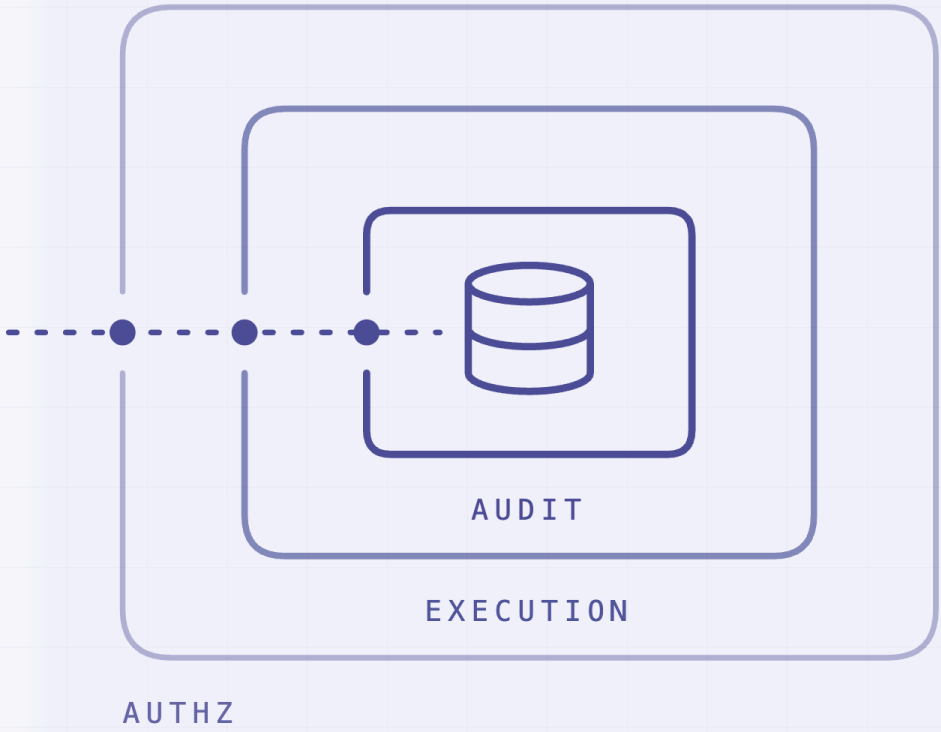
THE UNRELIABLE
SYSTEMS TRILOGY



Guide de l'accès aux bases de données en production

Muhammet Şafak

Guide de l'accès aux bases de données en production

Identités, credentials, accès just-in-time, exécution médiée et audit

Muhammet Şafak

© 2026 Muhammet Şafak
Téléchargement et lecture gratuits.

Table des matières

Préface 6

PARTIE I — FONDEMENTS **14**

Chapitre 1 — La porte que personne n'a conçue 15

Chapitre 2 — Un modèle de menaces pour les données de production 31

Chapitre 3 — Identités, rôles et moindre privilège 51

PARTIE II — L'ACCÈS DES APPLICATIONS **75**

Chapitre 4 — Credentials et secrets 76

Chapitre 5 — Les connexions comme politique 97

Chapitre 6 — Lectures, répliques et copies de données 117

PARTIE III — L'ACCÈS HUMAIN **132**

Chapitre 7 — Les chemins d'entrée : bastions, proxies et access gateways 133

Chapitre 8 — Accès just-in-time et break-glass 156

Chapitre 9 — Exécution médiée : analyse des requêtes et opérations dangereuses 183

Chapitre 10 — Requêtes ad hoc et corrections de données 201

Chapitre 11 — Changements de schéma et accès pour les migrations 227

PARTIE IV — PROTÉGER LES DONNÉES **250**

Chapitre 12 — Données sensibles : classification, masquage et minimisation 251

Chapitre 13 — Sauvegardes, exports et autres portes latérales 275

PARTIE V — PROUVER ET EXPLOITER **299**

Chapitre 14 — Audit et imputabilité 300

Chapitre 15 — L'accès pendant les incidents 320

Chapitre 16 — Une checklist d'accès et réflexions finales 340

Annexe 355

Préface

Imaginez un jeudi soir chez Ledgerly, une entreprise de facturation et de paiements qui n'existe pas. Robin est d'astreinte. Une facture client est bloquée à l'état « sending » depuis l'après-midi, et la correction paraît minime : une ligne, une colonne de statut. Le client de base de données de bureau de l'équipe mémorise le mot de passe admin partagé, se connecte au primary et fait un commit automatiquement, comme le font souvent ces clients. Une correction d'une ligne passe par ce client, sous un rôle qui peut modifier toutes les tables, et sa clause `WHERE` correspond à tout un lot de factures au lieu d'une seule.

Robin passe le reste de la soirée sur la réparation. Ce qui prend plus de temps, c'est la question qui suit : qui a exécuté cette instruction ? Le log de la base de données affiche le rôle admin partagé, qu'utilisent tous les services et, disons, vingt personnes. Le gestionnaire de mots de passe indique qui a ouvert l'entrée, pas qui a exécuté quoi. L'équipe peut reconstituer la modification à partir des effets de bord de l'application, et elle finit par le faire. Mais pendant une journée environ, personne ne peut dire avec assurance ce qui s'est passé exactement, sur quelles lignes, ni si quelque chose d'autre a utilisé la même porte au même moment.

Rien dans cette histoire n'était déraisonnable en soi. Le mot de passe était partagé parce que le partager était le moyen le plus rapide de faire tourner le premier service. Le client faisait des commits automatiques parce que c'est son comportement par défaut. Le rôle pouvait modifier toutes les tables parce qu'au début, c'était plus simple que de décider de quelles tables chaque service avait besoin. Chaque étape était raisonnable le jour où elle a été franchie. Ensemble, elles ont construit une porte que personne n'a conçue.

C'est de cette porte que parle ce livre.

Pourquoi ce livre existe

L'accès à une base de données de production est souvent traité comme de la plomberie. On le met en place au déploiement du premier service, puis on n'y retouche que lorsque quelque chose casse ou que quelqu'un s'en va. Pendant ce temps, la base de données devient ce que l'entreprise fait tourner de plus précieux : les clients, l'argent, l'historique. L'écart entre la dépendance d'une entreprise à son égard et le peu de conception consacré à son accès est le sujet de ce livre.

On a beaucoup écrit sur chaque pièce. La documentation des bases de données explique comment créer des rôles et des grants. Les outils de gestion des secrets expliquent comment émettre des credentials. Les recommandations de sécurité expliquent pourquoi l'accès permanent est un risque. Ce qui est plus difficile à trouver, c'est l'image d'ensemble vue du côté de l'ingénieur : comment les pièces s'emboîtent, ce que chacune coûte à ceux qui construisent et exploitent le système, et dans quel ordre une équipe au temps limité pourrait les adopter.

Ce livre essaie d'être cette image, et il part d'une affirmation. L'accès aux bases de données en production n'est pas un problème de distribution de credentials ; c'est un problème d'autorisation, d'exécution et d'imputabilité.

La vision par défaut traite l'accès comme une question de savoir qui détient un mot de passe et comment ce mot de passe est stocké, renouvelé et distribué. Le jeudi soir décrit plus haut montre où cette vision s'arrête. Robin détenait un credential légitime, et le credential a fait exactement ce pour quoi il avait été émis. Ce qui a échoué, c'est tout ce qui l'entourait : rien n'a décidé que cette personne devait exécuter une écriture sur cette table, rien n'a inspecté l'instruction ni borné son effet avant le commit, et rien n'a relié le résultat à une personne après coup. L'autorisation décide de ce qui peut se produire, et où. L'exécution, c'est la manière dont chaque instruction atteint les données. L'imputabilité, ce sont les preuves qui restent. Chaque chapitre nomme le pilier qu'il sert, et la Partie I construit le vocabulaire des trois. Le livre traite chaque contrôle comme une décision qui a un prix. Un contrôle adapté à la table des tokens de paiement peut être inutile pour une table de modèles de facture, et le livre le dira quand cela compte.

Je dois être clair sur la posture. Le livre est écrit du côté du défenseur. Les menaces ne sont décrites que dans la mesure où elles expliquent pourquoi un contrôle existe ; rien ici n'est un guide pour entrer. Il ne vend pas la peur et ne cite pas de statistiques de violations, parce qu'une revue de conception calme est plus utile qu'une revue alarmante. Et il compte la friction honnêtement. Chaque contrôle demande quelque chose aux ingénieurs et à la personne d'astreinte, et un contrôle qui ajoute de la friction sans réduire l'exposition relève du théâtre de sécurité. Quand je pense qu'une pratique répandue est du théâtre, je le dirai et j'expliquerai pourquoi.

À qui s'adresse ce livre

Le livre s'adresse aux **ingénieurs backend et plateforme** qui construisent des services qui parlent à une base de données de production, aux **SRE et DBA** qui exploitent cette base de données, aux **tech leads** qui doivent décider du fonctionnement de l'accès, et aux **ingénieurs devenus récemment responsables de la sécurité** dans une équipe qui n'a jamais eu de fonction dédiée. Si vous vous êtes déjà demandé si le mot de passe partagé était toujours partagé, si vous avez hésité avant d'accorder un accès en écriture à un collègue, ou si l'on vous a demandé « qui peut voir cette table ? » et que vous avez compris que la réponse honnête demandait une journée de recherches, vous êtes le lecteur que j'ai en tête.

Vous n'avez pas besoin d'une formation en sécurité. Vous devriez être à l'aise avec quelques notions :

- Les bases de données relationnelles : tables, rôles et grants, transactions, et ce qu'est une réplique.
- Les services qui se connectent à une base de données par le réseau, et le vocabulaire des connection pools et des timeouts.
- Le travail d'astreinte : être appelé par l'astreinte, réparer quelque chose sous pression, et consigner ensuite ce qui s'est passé.

Connaître un moteur de base de données particulier aide pour les exemples, mais n'est pas nécessaire, comme l'explique une section plus loin.

Ce que ce livre n'est pas

Ce n'est pas un livre d'administration de bases de données. Il ne vous apprendra pas à régler un planner, à dimensionner un buffer ni à choisir une topologie de réplication. Là où l'administration compte pour l'accès, comme pour les répliques et les migrations, le livre couvre cette partie et laisse le reste à la documentation de votre base de données.

Ce n'est pas un manuel de conformité. Des régimes comme le RGPD, la KVKK, PCI DSS, SOC 2 et HIPAA apparaissent parce qu'ils façonnent ce que demandent les auditeurs et les clients, et le livre ne les décrit que dans cet esprit : les régimes de ce type demandent généralement qui peut voir quoi, comment l'accès est approuvé, et quelle trace existe ensuite. Il n'interprète aucun régime, n'en cite pas les articles et ne vous dit pas ce qui est exigé de vous. Vos collègues des services juridique et conformité sont responsables de cette interprétation, et le but du livre est de leur donner une architecture sur laquelle ils peuvent raisonner.

Et ce n'est pas un livre d'attaque. Vous n'y trouverez pas de techniques d'intrusion. Quand un chapitre a besoin d'une menace, il nomme le risque en une phrase, comme une fuite de credential, une requête qui atterrit sur la mauvaise base de données ou une session de fournisseur restée ouverte, et passe au contrôle. Si décrire la menace plus précisément ne changeait rien à la conception, le détail est omis.

Comment le livre est organisé

Le livre compte cinq parties et seize chapitres, suivis d'une annexe.

La Partie I, Fondements (chapitres 1 à 3) défend la thèse ci-dessus et construit le vocabulaire pour la suite. Le Chapitre 1 examine comment les portes vers une base de données s'accumulent. Le Chapitre 2 construit un modèle de menaces pour les données de production qui inclut les erreurs autant que les abus. Le Chapitre 3 couvre les identités, les rôles et le moindre privilège, dont dépendent tous les chapitres suivants.

La Partie II, L'accès des applications (chapitres 4 à 6) porte sur les services. Le Chapitre 4 couvre les credentials et les secrets, des mots de passe statiques aux credentials dynamiques et à la workload identity. Le Cha-

pitre 5 traite les connexions comme une politique : pooling, limites et timeouts. Le Chapitre 6 couvre les lectures, les répliques et les autres copies de données que la production crée discrètement.

La Partie III, L'accès humain (chapitres 7 à 11) porte sur les personnes. Le Chapitre 7 compare bastions, proxies et access gateways. Le Chapitre 8 couvre l'accès JIT et break-glass. Le Chapitre 9 couvre l'exécution elle-même : comment une gateway inspecte chaque instruction, borne son effet et fait le commit. Le Chapitre 10 couvre les requêtes ad hoc et les corrections de données. Le Chapitre 11 couvre les changements de schéma et qui est autorisé à les faire.

La Partie IV, Protéger les données (chapitres 12 et 13) passe de qui peut se connecter à ce qu'on peut voir. Le Chapitre 12 couvre la classification, le masquage et la minimisation. Le Chapitre 13 couvre les sauvegardes, les exports et les autres portes latérales qui contournent tous les contrôles construits jusque-là.

La Partie V, Prouver et exploiter (chapitres 14 à 16) explique comment une équipe sait que ses contrôles fonctionnent et les maintient en état. Le Chapitre 14 couvre les logs d'audit et les revues des accès. Le Chapitre 15 couvre l'accès pendant les incidents, quand la pression est maximale et les raccourcis les plus tentants. Le Chapitre 16 propose une checklist, une échelle de maturité avec des tests de sortie et quelques réflexions finales sur la culture.

L'annexe rassemble un glossaire, une référence rapide des patterns, une matrice des opérations dangereuses, une correspondance de conformité et des modèles pour les revues des accès et les revues post-incident.

Chaque chapitre a la même forme. Il s'ouvre sur un court scénario ouvertement fictif. Il explique ensuite le problème et pourquoi l'approche évidente ne suffit pas, présente le contrôle, et en indique les coûts et les cas où ne pas l'utiliser. Il liste les erreurs courantes, traitées comme des pièges de conception et jamais comme des fautes personnelles, et se termine par cinq à sept points clés et une transition vers le chapitre suivant. Cette forme commune vous permet de lire les chapitres dans l'ordre ou de sauter à celui qui correspond au problème de la semaine, tout en trouvant la section des coûts là où vous l'attendez.

Faites la connaissance de Ledgerly

Ledgerly est un service fictif de facturation et de paiements interentreprises. Des entreprises émettent des factures par son intermédiaire, encaissent des paiements et tiennent un registre de ce qui a circulé. Il fait tourner cinq services : `billing-api`, `invoice-worker`, `payments-service`, `reporting-job` et `admin-console`. Sa base de données primary a deux répliques et une copie d'analytics pour l'équipe data. Elle stocke les coordonnées des clients, des informations de comptes bancaires et des tokens de paiement, ce qui explique pourquoi la conception de son accès compte.

Les personnes qui l'entourent occupent des rôles familiers. Il y a des ingénieurs backend d'astreinte, une équipe support, une équipe data, un petit groupe plateforme qui gère les bases de données, et un fournisseur comptable externe qui reçoit des exports.

Au début du livre, Ledgerly a un mot de passe admin partagé, un accès permanent pour la majeure partie de l'équipe backend, des clients de bureau sur les ordinateurs portables et aucune trace de qui a fait quoi. À la fin, il a un rôle par service, des credentials de courte durée, une gateway pour les humains, une manière relue de corriger les données, des instructions vérifiées et bornées avant de s'exécuter, des copies masquées et un log qui ne peut pas être réécrit en silence. Il y arrive une étape à la fois, et chaque chapitre indique quelle étape il ajoute et ce que cette étape coûte à l'équipe.

Ledgerly est inventé, tout comme tout ce qui lui arrive. Chaque chapitre s'ouvre sur un scénario signalé comme fictif, et tout chiffre qu'il contient est hypothétique. Des noms comme Robin, Sam, Jordan et Alex tiennent lieu de rôles au sein d'une équipe. Quand un scénario tourne mal, la leçon porte sur la conception qui l'a laissé mal tourner, et jamais sur la personne de l'histoire.

Comment lire le code

Aucun exemple de ce livre n'est écrit dans un vrai langage de programmation. Les procédures sont écrites en pseudocode simple, et les rôles, les grants et les tables utilisent une notation de type SQL qui n'appartient à aucun moteur particulier. Rien dans le livre n'est destiné à être collé dans

un terminal. Vous n'y trouverez pas de syntaxe de configuration pour un proxy, un vault ou un produit cloud précis, parce que cette syntaxe change alors que le pattern qui la sous-tend ne change pas.

Les diagrammes utilisent Mermaid. Chacun est précédé d'une phrase qui vous dit quoi regarder, et les chemins refusés ou en échec sont étiquetés. Les exemples ne contiennent jamais de secrets, de numéros de compte ou d'adresses e-mail réalistes ; des espaces réservés comme `<secret>` et `IBAN_PLACEHOLDER` les remplacent.

Pourquoi en anglais, et pourquoi indépendant du moteur

Le livre est écrit en anglais parce que l'essentiel du vocabulaire de ce domaine, du moindre privilège au break-glass, s'emploie en anglais même dans les équipes qui travaillent dans d'autres langues, et je voulais que les termes restent reconnaissables. Le livre est indépendant du moteur pour une raison voisine. Une équipe peut faire tourner PostgreSQL, MySQL ou autre chose, et peut utiliser un service d'identité cloud, un gestionnaire de secrets ou un script maison. Les patterns sont les mêmes pour tous : séparer le propriétaire des utilisateurs, donner à chaque workload sa propre identité, faire expirer les credentials, placer une entrée qui connaît l'identité devant les humains, vérifier ce qui s'exécute avant qu'il ne s'exécute, faire passer les écritures par quelque chose qui peut les refuser, enregistrer ce qui se passe, le revoir. Les produits et leur syntaxe changent souvent. Quand un produit réel est nommé, il l'est à titre d'exemple, et son comportement est décrit en des termes qui devraient rester valables d'une version à l'autre.

Une remarque sur la certitude

J'ai essayé d'indiquer lesquelles de mes recommandations sont fermes et lesquelles sont des préférences. Certaines sont fermes parce que le coût d'une erreur est l'exposition des données clients ou la perte du registre, et je le dis quand c'est le cas. La plupart relèvent du jugement, et une équipe réfléchie pourrait les pondérer autrement selon sa taille, ses régulateurs,

ses clients et la fréquence à laquelle l'astreinte est déclenchée. Quand je ne sais pas, je dirai « je ne sais pas » et je nommerai les variables qui trancheraient la question.

Si vous êtes en désaccord avec une recommandation, j'aimerais que vous puissiez trouver la prémisse avec laquelle vous êtes en désaccord. C'est pourquoi chaque chapitre expose son raisonnement et ses coûts. Une conception que vous comprenez et que vous avez adaptée à vos propres contraintes vaut plus qu'une conception copiée dans un livre.

Le Chapitre 1 commence par la porte elle-même.

PARTIE I

Fondements

Chapitre 1 — La porte que personne n'a conçue

Prenons Ledgerly dans ses premières années. Un client potentiel envoie un questionnaire de sécurité, et une ligne paraît anodine : « Listez chaque personne et chaque système qui peut se connecter à votre base de données de production. » Alex, qui s'occupe des bases de données en plus d'autres tâches, s'en charge par un après-midi calme.

Alex commence par le gestionnaire de mots de passe, qui contient une seule entrée admin que toute l'équipe backend connaît. Viennent ensuite les cinq services, qui stockent chacun ce même mot de passe, et les ordinateurs portables, où les clients de base de données de bureau le mémorisent. Puis la copie d'analytics, une restauration nocturne d'un dump non masqué, et quelques dumps plus anciens dans un stockage partagé dont personne ne se rappelle la création. Chaque endroit qu'Alex vérifie mène à un autre.

Le questionnaire a une deuxième ligne : « Comment savez-vous ce qu'ils ont fait ? » Alex ouvre le log de la base de données et trouve un seul nom de rôle sur chaque ligne, que l'instruction vienne d'un service, d'un ingénieur qui corrigeait une ligne ou d'un rapport qui aurait dû lire une réplique. La réponse honnête à la première question est un tas d'endroits, et la réponse honnête à la seconde est que Ledgerly ne sait pas.

Alex n'a pas besoin d'un meilleur mot de passe. Alex a besoin de savoir de quoi l'accès est fait, et à quoi ressemblerait une réponse courte et défendable. C'est la question de ce chapitre : de quoi est fait l'accès à une base de données de production, et que faudrait-il à Ledgerly pour répondre à ce questionnaire sans une semaine de recherches ?

Ce chapitre décrit l'état initial de Ledgerly, comme le fait le Chapitre 2, et n'ajoute encore aucune étape vers l'état cible. Son rôle est de vous donner le vocabulaire et la carte des étapes qui suivent.

Le problème, et pourquoi il est difficile

L'accès à une base de données de production est rarement conçu. Il s'accumule. Le premier service a besoin d'une entrée, alors quelqu'un crée un rôle admin et un mot de passe. Le deuxième service le réutilise parce que le premier a fonctionné. Un ingénieur doit regarder une ligne, alors le mot de passe entre dans un client de bureau. Un job planifié a besoin de données et reçoit la même connection string que tout le reste. Aucune de ces étapes n'est une décision sur l'accès, et leur somme est la porte que nomme le titre du chapitre.

Regardez à quoi ressemble la porte de Ledgerly au moment où Alex se met à compter. Le mot de passe admin partagé se trouve dans un gestionnaire de mots de passe d'équipe, et supposons, à titre d'illustration, qu'une vingtaine de personnes le connaissent. Le rôle admin qu'il déverrouille est propriétaire de toutes les tables. Les cinq services se connectent avec ce rôle, en utilisant le même mot de passe stocké dans la configuration de chaque service, et le pipeline de déploiement exécute aussi les migrations avec lui. Les ingénieurs atteignent le primary depuis leurs ordinateurs portables via le VPN de l'entreprise, avec des clients qui mémorisent le mot de passe et font un commit de chaque instruction dès qu'elle s'exécute. L'accès est permanent : personne ne le perd jamais, y compris les personnes passées dans une autre équipe il y a un an.

Les répliques acceptent le même login admin, et des requêtes de reporting s'exécutent parfois sur le primary. La copie d'analytics est une restauration d'un dump non masqué, et les fichiers de dump traînent. Un fournisseur comptable externe ne se connecte jamais à aucune base de données de Ledgerly, mais il reçoit des exports de données de factures et du registre, et l'étape d'export de `reporting-job` écrit ces fichiers avec le login admin partagé. Le log de la base de données affiche un seul rôle. Rien dans cet agencement n'enregistre quelle personne a exécuté quelle instruction.

Il est tentant de traiter cela comme un problème de discipline : dire aux gens d'être prudents, changer le mot de passe plus souvent. Je ne commencerais pas par là. Une règle qui ne fonctionne que lorsque tout le monde s'en souvient échoue la première fois que la personne d'astreinte est fatiguée. Le système donne à chaque principal les mêmes capacités et

le même nom, si bien qu'une personne qui veut être prudente n'a rien avec quoi l'être : aucun rôle plus étroit à choisir, aucun moyen de dire « lecture seule » à la base de données, aucune session qui porte un nom.

Il en va de même pour le remède auquel on pense en premier, qui consiste à mieux gérer le mot de passe : le stocker dans un vault, le renouveler, ne le donner qu'à ceux qui le demandent. C'est de la distribution de credentials, et cela vaut la peine d'être fait, mais cela ne répond qu'à une question : qui détient le secret. L'accès aux bases de données en production n'est pas un problème de distribution de credentials ; c'est un problème d'autorisation, d'exécution et d'imputabilité. L'autorisation demande ce que chaque principal peut faire, et par quelle voie. L'exécution demande comment chaque instruction atteint les données, si quelque chose l'inspecte d'abord, et jusqu'où elle peut aller. L'imputabilité demande si les preuves relient chaque instruction à une personne et à une demande. Un mot de passe parfaitement stocké règle la question de savoir qui détient le secret, et rien d'autre : la personne qui le détient peut toujours exécuter une instruction que personne n'a inspectée, et le log peut toujours afficher un nom partagé.

C'est pourquoi il s'agit d'une question d'ingénierie plutôt que de politique. Une politique énonce ce qui devrait se produire. Une conception détermine ce qui peut se produire, et ce qui laisse une trace quand cela se produit. La suite de ce chapitre prend la forme d'un problème de conception : en quoi consiste l'accès, ce que nous voulons qu'il fasse, et où chaque partie est traitée dans ce livre.

L'accès se compose de cinq parties

Avant de juger une conception, il est utile d'avoir une définition précise de ce que l'on conçoit. Je trouve utile de dire que l'accès à une base de données de production comporte cinq parties : une identité, un grant, un chemin, une exécution et un enregistrement. Le tableau indique à quelle question répond chaque partie et à quel pilier de la thèse du livre elle appartient.

Partie	Question	Pilier
Identité	Qui ou quoi agit : une personne ou une instance de workload ?	Imputabilité

Partie	Question	Pilier
Grant	Que ce principal peut-il faire, tout court : rôles, privilèges, plafond ?	Autorisation
Chemin	Comment la connexion arrive-t-elle : accès réseau, gateway, pooler ?	Autorisation (où)
Exécution	Comment une instruction atteint-elle les données ? Qui ou quoi la soumet, est-elle inspectée avant de s'exécuter, jusqu'où peut-elle porter (transaction, row guard, timeout), et comment fait-on son commit.	Exécution
Enregistrement	Quelles preuves restent, et relient-elles l'instruction à la personne, à la demande et au résultat ?	Imputabilité

Les trois piliers se répartissent sur les parties comme suit. L'autorisation, c'est le grant plus le chemin, décidés par demande. L'exécution, c'est ce qui arrive à chaque instruction. L'imputabilité, c'est l'identité plus l'enregistrement. Un mot de passe ne touche que le premier de ces piliers, et seulement à la porte.

L'**identité** est l'affirmation vérifiée qu'un principal est bien ce qu'il prétend être. Un **principal** est tout ce qui s'authentifie : une personne, un service, un job planifié, une intégration de fournisseur. Principal est le mot à employer quand un énoncé vaut aussi bien pour les humains que pour les workloads, et ce livre l'emploie ainsi d'un bout à l'autre. Un rôle n'est pas une identité. Un **rôle** est un ensemble nommé de privilèges à l'intérieur de la base de données, et il peut être partagé par de nombreux principals, ce que fait précisément le rôle admin partagé.

Le **grant** est ce que le principal est autorisé à faire une fois entré. Ici, un grant est l'acte, et l'enregistrement qui en résulte, d'attribuer un privilège à un rôle, ou un rôle à un **login** ; le Chapitre 3 élargit la définition. Un **privilège** est une chose précise, comme lire une table, et le login est le compte au niveau de la base de données avec lequel un principal se connecte. Le **chemin** est la route qu'emprunte une connexion pour atteindre la base de données : un réseau, un proxy, une gateway, un ordinateur portable.

L'**exécution** est la manière dont une instruction atteint les données : qui ou quoi la soumet, si quelque chose l'inspecte avant qu'elle ne s'exécute, ce qui borne son effet, et comment on fait son commit. Un grant dit qu'un rôle peut mettre à jour `invoices`. Il ne dit pas que cette mise à jour doit

toucher une seule ligne, que quelqu'un l'a regardée d'abord, ni qu'elle s'exécute dans une transaction qui peut faire l'objet d'un rollback. Dans l'état initial, rien ne répond à ces questions : le client de bureau soumet tout ce qui est tapé et en fait le commit sur-le-champ. Le Chapitre 9 porte sur cette partie.

L'**enregistrement** est la preuve après coup, la réponse à « qui a fait quoi, et quand ? », et, pour une écriture, « sur quelles lignes, dans le cadre de quelle demande ? »

Chaque partie peut échouer seule, et pourtant chacune dépend des autres pour être utile. Sans identité distincte, un grant précis n'a personne à qui se rattacher. Sans chemin choisi, une identité peut être contournée par une route qui n'en demande jamais. Sans exécution contrôlée, un grant étroit autorise encore une instruction plus large que ce que quiconque voulait. Sans enregistrement, vous ne pouvez pas savoir si les quatre autres ont fonctionné. Cette dépendance explique pourquoi corriger une seule partie, généralement le mot de passe, laisse le reste du problème intact.

Un test: le log peut-il répondre à la question?

Une façon de découvrir quelles parties manquent est de poser une question au log et d'observer où la réponse s'arrête. Le pseudocode suivant suit une seule instruction à travers les cinq parties, en n'utilisant que ce que l'état initial enregistre.

```
function explainStatement(logLine):
    login = logLine.login
    // identity: does this login map to exactly one principal?
    principals = principalsUsing(login)
    if count(principals) != 1:
        return { answer: "unknown", stoppedAt: "identity" }
    // path: did the connection arrive by a route we chose?
    if not isKnownPath(logLine.source):
        return { answer: "unknown", stoppedAt: "path" }
    // grant: was this statement within what the principal was meant
to do?
    if not withinIntendedGrant(principals[0], logLine.statement):
        return { answer: principals[0], stoppedAt: "grant" }
    // execution: was a write reviewed and bounded before it
committed?
    if isWrite(logLine.statement):
        if not logLine.reviewedStatement or not
logLine.countShownBeforeCommit:
            return { answer: principals[0], stoppedAt: "execution" }
        if logLine.rowsAffected is unknown or logLine.rowsAffected >
logLine.expectedRows:
            return { answer: principals[0], stoppedAt: "execution" }
    // record: does the evidence tie the statement to the person, the
request and the result?
    if logLine.correlationId is empty or logLine.result is unknown:
        return { answer: principals[0], stoppedAt: "record" }
    return { answer: principals[0], stoppedAt: null }
```

Dans l'état initial de Ledgerly, `principalsUsing("admin")` renvoie une vingtaine de personnes, cinq services et le pipeline de déploiement, si bien que la fonction s'arrête au premier contrôle pour chaque instruction. Ce n'est pas un bug du log. Le log dit la vérité sur ce qu'il a vu, à savoir un seul login. La lacune est dans la conception qui a placé de nombreux principals derrière un seul login. Corrigez l'identité, et la même fonction s'arrêterait ensuite au contrôle de l'exécution. L'incident fondateur est le cas pour lequel ce contrôle existe : une correction d'une ligne dont la clause `WHERE` correspond à tout un lot a un `rowsAffected` très supérieur à `expectedRows`, n'a jamais été relue et n'a jamais affiché son nombre de lignes avant `COMMIT`, si bien qu'elle s'arrête à « execution » alors même que la personne est connue et que le grant l'autorisait. Une instruction qui passe ce contrôle peut encore s'arrêter à « record » si rien ne la relie à une demande et à un résultat.

Applications et personnes, une seule question

Quand on entend « accès à la base de données », on imagine souvent des ingénieurs avec des outils de requête. Cette image est incomplète, et d'une manière qui compte. Une grande partie du trafic vers le primary de Ledgerly vient de logiciels : `billing-api` qui sert l'application web, `invoice-worker` qui envoie des relances, `payments-service` qui dialogue avec les prestataires de paiement, `reporting-job` qui tourne la nuit, et `admin-console` qui agit pour le personnel du support. Les logiciels se connectent plus souvent, plus longtemps et avec moins de supervision que n'importe quelle personne.

Les services et les personnes échouent de manières différentes, c'est pourquoi la Partie II couvre les applications et la Partie III les personnes. Le contraste est concret :

	Services	Personnes
Fréquence	Constante, jour et nuit	Occasionnelle, par à-coups
Supervision	Aucune pendant l'exécution	Une personne surveille le résultat
Durée de vie du credential	Courte, émis par un broker (Chapitre 4)	Une session ou une fenêtre approuvée (chapitres 7 et 8)
Défaillance typique	Un secret qui fuit, un connection pool épuisé, un déploiement qui pointe vers la mauvaise base de données	Une instruction plus large que voulu, exécutée à la main sous pression

Les deux reposent sur les fondements posés dans la Partie I.

Il existe un autre type de principal facile à oublier : un service qui agit pour le compte d'une personne. `admin-console` est utilisé par le personnel du support, mais dans l'état initial il se connecte avec le login admin partagé comme tout le reste, si bien que la base de données voit la console, jamais la personne qui a cliqué. Sam, au support, consulte un client à la fois, et la conception permet à la console d'atteindre les données de tous les clients sous un nom qui n'est celui de personne. Le Chapitre 3 montre comment la console transmet l'identité de la personne avec chaque action, et le Chapitre 14 comment les enregistrements d'audit sont joints, afin que la personne survive au saut.

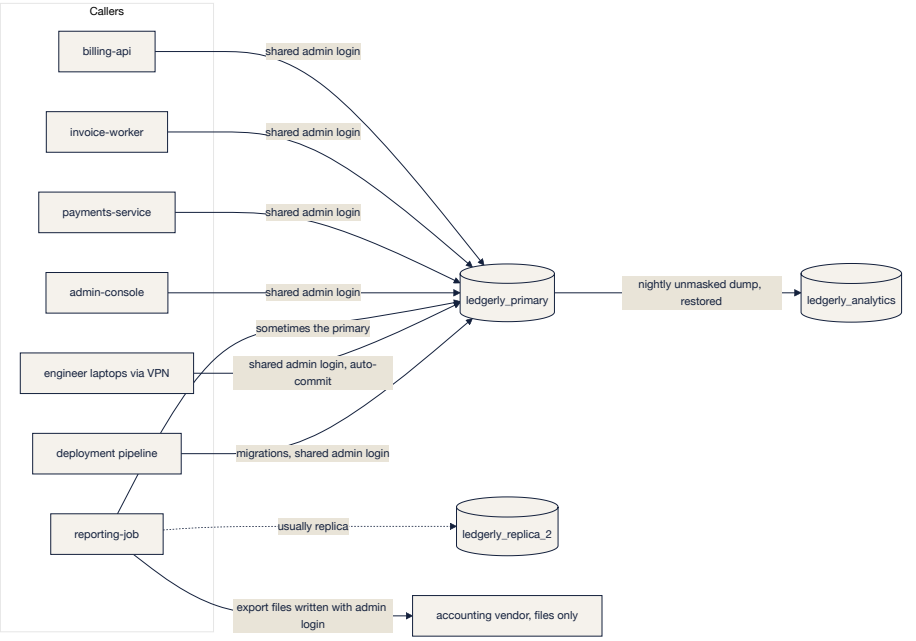
Sécurité et fiabilité sont une même question

Les équipes tiennent souvent deux listes. L'une porte sur la sécurité : qui pourrait voler des données, qui pourrait abuser de l'accès. L'autre porte sur l'exploitation : ce qui pourrait faire tomber la base de données ou la corrompre. Elles appartiennent généralement à des personnes différentes et sont revues à des moments différents. Pour l'accès, je les fusionnerais, parce que les questions sont les mêmes.

Prenez trois événements qui semblent sans rapport. Une instruction avec une clause `WHERE` erronée ou absente met à jour toutes les factures au lieu d'une seule. Un client sur un ordinateur portable laisse une connexion ouverte et occupe un slot toute la soirée. Un dump de la base de données de production est copié dans un stockage partagé et lu par quelqu'un qui n'avait rien à y lire. Le premier est un problème d'intégrité des données, le deuxième un problème de disponibilité, le troisième un problème de confidentialité. Pourtant, chacun pose la même question à la conception : pourquoi un seul principal a-t-il pu faire quelque chose d'aussi large, sans route plus étroite, sans limite ni enregistrement ?

Les trois événements partagent une cause de conception : aucun rétrécissement du grant, de la durée de vie ou du chemin, et rien à montrer après coup. La mauvaise clause `WHERE` est surtout une lacune de grant et d'exécution, la connexion ouverte une lacune de durée de vie et de limite, et le dump copié une lacune de chemin. Rétrécir le grant, limiter la durée de vie, choisir le chemin, borner l'instruction et conserver l'enregistrement aident pour les trois événements à la fois. C'est pourquoi une seule conception sert les deux listes. Un rôle qui ne peut pas lire `payment_tokens` protège les clients d'une fuite, et un rôle qui ne peut pas écrire dans `ledger_entries` protège le registre d'une mauvaise mise à jour par lot. Les deux viennent de la même habitude d'accorder des grants étroits.

Le diagramme ci-dessous montre l'état initial de Ledgerly. Regardez combien de flèches aboutissent au même nœud, et la flèche qui en part vers la copie d'analytics.



Deux choses ressortent. D'abord, chaque chemin vers le primary porte le même nom, si bien que la base de données ne peut pas distinguer un worker d'une personne. Ensuite, les données sortent aussi par une route latérale : la restauration nocturne copie tout, y compris des colonnes dont aucun analyste n'a besoin, vers un stockage qui a son propre public.

L'éventail des voies d'entrée

Entre le mot de passe partagé et une conception entièrement contrôlée s'étend un éventail de patterns, et les chapitres suivants placent chacun d'eux sur cet éventail. Le diagramme va du moins au plus de contrôle sur chaque instruction. Regardez ce qu'ajoute chaque étape, et où apparaît le premier contrôle au niveau de l'instruction.



Chaque étape vers la droite ajoute une propriété et coûte quelque chose. Un bastion restreint le chemin, mais la session de l'autre côté reste sans supervision. Un session proxy relie une connexion à une personne et l'enregistre, mais il remet une session dans laquelle tout ce que le grant permet peut être tapé. Une gateway qui assure la médiation de l'exécution

examine chaque instruction avant qu'elle ne s'exécute et peut la refuser. Un chemin produit supprime complètement la session : la personne demande une opération définie, comme « renvoyer cette facture », et n'écrit jamais de SQL. Rien ici ne dit que le pattern le plus à droite est toujours le meilleur ; c'est aussi le plus coûteux à construire et le plus lent à faire évoluer. Le Chapitre 7 compare les quatre premiers sur des chemins concrets, le Chapitre 9 couvre les contrôles au niveau de l'instruction, et le chapitre de clôture note les patterns par pilier.

À quoi ressemble un bon accès

J'écrirais l'objectif avant de choisir le moindre outil. J'utilise cinq propriétés, et chacune a un prix.

Le **moindre privilège** est la pratique qui consiste à n'accorder que ce dont une tâche a besoin, pour la durée dont elle a besoin. Le prix est l'effort nécessaire pour découvrir ce dont chaque tâche a besoin, et la friction quand une tâche s'avère en demander davantage. Une équipe qui l'adopte sans moyen rapide de demander plus verra les gens le contourner.

La **durée de vie courte** signifie que l'accès prend fin de lui-même. Un grant qui dure une heure n'a pas besoin que quelqu'un s'en souvienne. Comparez avec l'**accès permanent**, qui existe en continu, qu'il soit utilisé ou non. Un grant permanent est la promesse que personne n'aura besoin de redemander, et il est commode le premier mois et coûteux tous les mois suivants, parce que personne ne se rappelle ce que couvrait la promesse. Le coût de la durée de vie courte, c'est de devoir demander plus souvent, et une expiration qui peut tomber au milieu d'une réparation.

L'**observabilité** signifie que chaque usage de l'accès laisse un enregistrement qui nomme le principal, et que cet enregistrement se trouve quelque part où le principal ne peut pas le modifier. Le coût est le stockage et, surtout, l'attention : un log que personne ne lit relève du théâtre de sécurité, puisqu'il coûte du stockage et n'apporte aucune protection. Le Chapitre 14 y revient.

La **réversibilité** a deux sens, et les deux comptent. L'accès doit être facile à retirer, et une erreur commise avec un accès doit pouvoir être annulée ou au moins contenue. La révocation doit être une routine, et annuler une erreur demande des corrections relues et des sauvegardes qui se res-

taurent. Une forme peu coûteuse de réversibilité consiste à exécuter une écriture dans une transaction explicite et à afficher le nombre de lignes avant `COMMIT` : une mise à jour censée toucher une ligne et qui en annonce, disons, quatre cents est arrêtée tant qu'un rollback est encore possible. Le Chapitre 9 explique comment une gateway en fait le comportement par défaut. Le coût est le temps passé en répétitions.

La **faible friction** est celle que les équipes oublient d'écrire. Si le chemin légitime est lent, les gens trouveront le chemin illégitime, et ils auront raison de le faire, parce que l'incident de 2 h du matin n'attend pas. Le coût de la faible friction est qu'elle contraint les quatre autres : chaque contrôle doit être assez peu coûteux pour être utilisé pendant la pire heure de la semaine.

Ces cinq propriétés tirent les unes contre les autres. La durée de vie courte ajoute de la friction. L'observabilité ajoute un saut : une session enregistrée doit passer par quelque chose qui l'enregistre, et ce quelque chose peut être lent ou en panne. Le moindre privilège multiplie les rôles. Je ne pense pas qu'il existe un réglage qui maximise les cinq. La question utile n'est pas « qui devrait avoir accès ? » mais « quelles instructions peuvent s'exécuter, par quoi, et avec quel enregistrement ? » Chaque chapitre suivant répond à une part de cette question, et dit ce que la réponse coûte.

Un premier inventaire

Avant d'adopter le moindre contrôle, j'écrirais ce qui existe. L'esquisse suivante montre la forme d'un inventaire pour l'état initial de Ledgerly. L'idée est d'avoir une ligne par porte, avec un responsable nommé et une raison, afin qu'une ligne sans responsable apparaisse comme une lacune.

```
door:        desktop client on engineer laptops
principals:  about twenty people (illustrative)
login:       shared admin
reaches:     ledgerly_primary, ledgerly_replica_1, ledgerly_replica_2
network source: company VPN, any laptop
grants:      owns every table
can write?:  yes, auto-commit, no row limit
record:      database log shows the shared role only
owner:       nobody
reason:      "needed to fix rows during incidents"
ends:        never
```

Remplissez un bloc par porte : chaque service, le job planifié, le pipeline de déploiement, les ordinateurs portables, la restauration d'analytics, les dumps dans le stockage partagé et les fichiers d'export. La plupart des équipes constatent que ce sont les lignes `owner` et `ends` qui restent vides. Un inventaire comme celui-ci n'est pas un plan de projet. C'est ce qui transforme « qui peut atteindre le primary ? » d'une journée de recherches en un document que vous pouvez lire.

Une carte de ce livre

Les chapitres qui suivent abordent tour à tour chaque partie de l'accès et chaque défaillance décrite ci-dessus. Le tableau relie chaque question soulevée par ce chapitre à l'endroit où le livre y répond.

La question	Pilier	Où elle trouve sa réponse
Contre qui et quoi protégeons-nous les données, et contre quels types d'événements ?	Les trois	Chapitre 2
Comment donner à chaque principal une identité distincte et le rôle le plus étroit qui fonctionne ?	Autorisation	Chapitre 3
Comment les services obtiennent-ils des credentials sans mot de passe partagé dans un fichier ?	Imputabilité (identité)	Chapitre 4
Comment les limites de connexion et les timeouts deviennent-ils une politique qui protège la base de données ?	Autorisation et exécution (bornes)	Chapitre 5
Comment les lectures atteignent-elles les répliques, et qu'advient-il des autres copies de données ?	Autorisation	Chapitre 6
Par quels chemins les personnes atteignent-elles la base de données, et que trouve-t-on sur chaque chemin ?	Autorisation (chemin), imputabilité	Chapitre 7
Comment quelqu'un obtient-il un accès en écriture pour une tâche, et comment cet accès prend-il fin ?	Autorisation	Chapitre 8
Comment une instruction est-elle vérifiée avant de s'exécuter, et comment fait-on son commit ?	Exécution	Chapitre 9
Comment les requêtes ad hoc et les corrections de données restent-elles relisibles ?	Exécution	Chapitre 10

La question	Pilier	Où elle trouve sa réponse
Qui modifie le schéma, et avec quel accès ?	Exécution	Chapitre 11
Quelles données exigent une protection supplémentaire, et comment sont-elles masquées ou retenues ?	Autorisation	Chapitre 12
Qu'en est-il des sauvegardes, des exports et des autres routes qui contournent les contrôles ?	Autorisation (chemin)	Chapitre 13
Comment conserver un enregistrement qui tient la route, et le revoir ?	Imputabilité	Chapitre 14
Qu'est-ce qui change pendant un incident ?	Les trois, sous pression	Chapitre 15
Que faisons-nous d'abord, et comment le maintenons-nous ?	Les trois	Chapitre 16

Le livre compte cinq parties : les fondements, l'accès des applications, l'accès humain, la protection des données et de leurs copies, et la preuve et l'exploitation de l'ensemble. Chaque chapitre nomme dans son ouverture le pilier qu'il sert. Vous n'êtes pas obligé de lire dans l'ordre. Si votre problème immédiat est le mot de passe partagé dans cinq fichiers de configuration, le Chapitre 4 est le point de départ, mais le Chapitre 3 vous évitera de créer par accident le prochain rôle partagé.

Compromis et coûts

Le premier coût est **le temps et l'attention d'un petit groupe**. Quelqu'un doit construire l'inventaire, définir les rôles, mettre en place les credentials et mener les revues. Dans une petite équipe, c'est souvent le travail à temps partiel d'une seule personne, et il entre en concurrence avec le travail produit. Si ce travail n'est jamais planifié, la conception reste sur le papier. Je préfère voir une première étape petite et terminée qu'un grand plan.

Le deuxième coût est **la friction pour la personne d'astreinte**. Chaque contrôle place une étape entre une personne fatiguée et une réparation. Des rôles étroits signifient que la première tentative échoue parfois avec une erreur de privilège au pire moment. Un accès de courte durée signifie

qu'il faut redemander. Décidez à l'avance quel est le chemin rapide, et répétez-le. Un contrôle contourné à chaque incident n'est pas un contrôle ; c'est un délai avant le contournement.

Le troisième coût est **de nouveaux systèmes qui doivent être disponibles quand on en a besoin**. Une gateway, un stockage de secrets et un stockage de logs sont chacun une chose de plus qui peut tomber en panne. Avant d'en adopter un, demandez-vous ce qui arrive à la personne qui a besoin de la base de données pendant cette panne, et gardez une réponse prête.

Il existe aussi des cas où la conception complète n'est pas le bon point de départ. Un prototype à un seul service sans données clients, géré par deux personnes, n'a besoin ni d'une gateway ni d'un calendrier de revues. Il a besoin en revanche de son propre login, distinct de celui de toute personne, parce que cela ne coûte pas cher, et d'une note indiquant ce qui ferait revenir l'équipe sur la question : le premier client, le troisième ingénieur ou la première intégration externe. Je ne connais pas le bon moment pour votre entreprise. Cela dépend, et voici de quoi : de la sensibilité des données, du nombre de principals qui y touchent, et de la vitesse à laquelle une mauvaise instruction peut causer des dégâts irréversibles.

Enfin, les propriétés d'un bon accès ont une limite. Elles réduisent le rayon d'impact et rendent les comportements visibles. Elles ne rendent pas une base de données sûre, et aucun chapitre de ce livre ne prétend le contraire. Le **rayon d'impact** est l'étendue de ce qui peut être affecté si un principal, un credential ou une action tourne mal, et le travail de ce livre consiste à rendre ce rayon plus petit et plus visible, non à le réduire à zéro.

Erreurs courantes

Ce sont des pièges de conception, pas des fautes personnelles. Chacun est tentant pour une raison, et chacun a un remède qui ne dépend pas de la prudence accrue de qui que ce soit.

- **Traiter un changement de mot de passe comme le remède.** Un nouveau mot de passe donne une impression de progrès, et il efface ce que savait un ancien collègue. Mais si, disons, vingt personnes et cinq services le partagent toujours, la conception est inchangée. La correc-

tion est une conception dans laquelle personne n'a besoin de partager un mot de passe, et c'est ce que construisent les chapitres 3 et 4.

- **Corriger le grant et oublier le chemin.** Les rôles étroits n'aident que si les connexions sont obligées de les utiliser. Si les ordinateurs portables peuvent encore atteindre le primary avec un ancien login admin, les rôles étroits sont facultatifs. Inventoriez chaque chemin, pas seulement chaque rôle.
- **Oublier les applications.** Les revues de « qui a accès à la base de données » ne listent souvent que des personnes, parce que les personnes figurent sur un organigramme. Les services détiennent les credentials les plus puissants et les utilisent davantage. Incluez chaque workload, chaque job planifié et chaque script.
- **Rendre le chemin sûr plus lent que le chemin risqué.** Si lire la production demande un ticket et une journée alors que le mot de passe partagé prend, disons, dix secondes, le mot de passe partagé gagne à chaque fois. Visez un chemin sûr qui ne soit pas plus lent.

> **Avertissement :** Un dump non masqué dans un stockage partagé est la voie la plus directe vers les données clients dans l'état initial, et il est ouvert à quiconque peut lire le dossier. Considérez la recherche et la suppression de ces fichiers comme faisant partie du premier inventaire, avant toute refonte des rôles.

Points clés

- L'accès aux bases de données en production n'est pas un problème de distribution de credentials ; c'est un problème d'autorisation, d'exécution et d'imputabilité, et un mot de passe ne répond qu'à la première question, à la porte. L'accès est une conception en cinq parties, à savoir Identité, Grant, Chemin, Exécution et Enregistrement, et une faiblesse dans l'une d'elles fragilise les autres.
- Un compte admin partagé n'est pas une décision que quelqu'un a prise ; il s'accumule, et le remède est une conception dans laquelle personne n'a besoin de partager un mot de passe.
- Les applications se connectent plus souvent et avec moins de supervision que les personnes, donc tout traitement de l'accès doit couvrir les

services, les jobs et les outils comme des principals, aux côtés des humains.

- La sécurité et la fiabilité opérationnelle sont une même question pour l'accès, parce qu'une clause `WHERE` erronée, une connexion qui fuit et un dump copié partagent une cause de conception : aucun rétrécissement du grant, de la durée de vie ou du chemin, et aucun enregistrement.
- Un bon accès a cinq propriétés, à savoir le moindre privilège, la durée de vie courte, l'observabilité, la réversibilité et la faible friction, et elles tirent les unes contre les autres, si bien que chaque contrôle doit être présenté avec ce qu'il coûte.
- La première étape est un inventaire avec une ligne par porte, y compris les services, les ordinateurs portables et les copies de données, où un responsable manquant ou une date de fin manquante apparaît comme une lacune visible.
- Les contrôles réduisent le rayon d'impact et rendent les comportements visibles, mais ils ne rendent pas une base de données sûre, et l'objectif honnête est un rayon plus petit et une vue plus claire.

Le chapitre suivant passe de la forme de l'accès aux raisons de le protéger. Le Chapitre 2 demande pour qui et pour quoi existent les contrôles, classe en trois types les événements qui peuvent mal tourner, et rédige pour Ledgerly un modèle de menaces auquel le reste du livre pourra se référer.

Chapitre 2 — Un modèle de menaces pour les données de production

Imaginez que le contact sécurité de Ledgerly envoie une courte question à l'équipe plateforme avant une revue trimestrielle : « Contre qui protégeons-nous la base de données de production ? » Alex, qui gère les bases de données, répond en moins d'une minute : contre les personnes extérieures, surtout. Puis Alex marque une pause, parce que la réponse ne correspond pas aux propres tickets de l'équipe sur l'année écoulée. Une demande de support qui a touché le mauvais client. Une requête de reporting qui a ralenti les écritures de factures pendant tout un après-midi. Un dump restauré sur un ordinateur portable pour une journée de débogage, que personne n'a pensé à rappeler de supprimer. Un export vers un fournisseur parti avec une colonne de plus que prévu.

Aucun de ces cas n'impliquait une personne extérieure. Aucun n'impliquait non plus de mauvaise intention. Tous impliquaient une porte ouverte à quelqu'un qui avait une tâche raisonnable, et un rôle qui permettait davantage que ce dont la tâche avait besoin. La première réponse d'Alex était celle qui sonne comme de la sécurité. La seconde liste est celle qui décrit le système.

Ce chapitre décrit Ledgerly dans son état initial, comme l'a fait le Chapitre 1, et n'ajoute aucun contrôle qui lui soit propre. Ce qu'il ajoute, c'est un modèle de menaces écrit, la référence à l'aune de laquelle chaque contrôle ultérieur du livre est choisi. Il situe les menaces par rapport aux trois piliers : ce qu'un principal est autorisé à faire, comment chaque instruction atteint les données, et quelle preuve subsiste. La question à laquelle il répond est celle à laquelle Alex n'a pas pu répondre en une minute : qui et quoi peut nuire aux données, par quelles portes, et jusqu'où les dégâts se propagent-ils ?

Le problème, et pourquoi il est difficile

Une conception des accès sans modèle de menaces tend à suivre l'inquiétude la plus bruyante. Si c'est un attaquant externe, l'équipe construit un périmètre : un VPN, une règle de pare-feu, une allowlist réseau. Ces éléments comptent. Mais un périmètre répond à « qui peut s'approcher de la base de données ? » et ne dit rien de ce qu'une personne ou un service peut faire une fois à l'intérieur. Dans l'état initial de Ledgerly, quiconque est sur le VPN avec le mot de passe partagé peut faire tout ce que peut faire le rôle admin, c'est-à-dire tout.

L'alternative évidente est une liste de contrôles tirée d'une checklist, appliqués dans l'ordre. Une checklist ne sait pas lesquelles de vos portes portent le plus de risque. Elle demandera la rotation d'un mot de passe que personne ne pourrait faire tourner sans interruption de service, et ne dira rien de la restauration nocturne qui copie chaque colonne Restricted vers un stockage dont l'audience est bien plus large. Des contrôles choisis sans modèle sont choisis selon la disponibilité et la mode. Certains réduisent l'exposition. D'autres relèvent du théâtre de sécurité, une friction sans réduction correspondante du risque, et l'équipe ne peut pas distinguer les uns des autres parce qu'elle n'a jamais écrit à quoi servait le contrôle.

Un modèle écrit est nécessaire pour trois raisons. D'abord, les portes sont des objets techniques : un login, un chemin réseau, un fichier dans un stockage partagé, une variable de pipeline. Une politique peut dire « l'accès doit être approprié », mais seule une conception décide de ce qu'un login peut atteindre. Ensuite, les plus grands risques viennent de l'interaction de choix ordinaires, comme le décrit la préface : un mot de passe partagé, un client qui fait un commit automatique et un rôle propriétaire de toutes les tables. Aucun choix isolé ne paraît mauvais, si bien qu'une revue des choix un par un ne trouvera pas le problème. Enfin, le modèle doit être assez peu coûteux à maintenir, sinon il sera écrit une fois puis laissé à la dérive.

Un modèle de menaces, au sens où on l'entend ici, est petit. Il liste qui et quoi peut agir sur les données, par quel chemin, ce qui pourrait mal tourner, jusqu'où l'effet se propage, et quel contrôle le traite. Ce n'est ni un score de risque ni une prédiction. C'est une manière de poser la même question pour chaque porte, afin que les réponses puissent être comparées.

> **Remarque :** Le livre décrit les menaces seulement dans la mesure où elles justifient un contrôle. Ce chapitre nomme ce qui peut mal tourner, comme la fuite d'un credential ou une requête qui atterrit sur la mauvaise base de données, et ne décrit pas comment quiconque provoquerait cela. Si davantage de détails ne changeraient pas la conception, ils sont omis.

Qui et quoi atteint la base de données

Le Chapitre 1 a dit que l'accès à une base de données de production comporte cinq parties : une Identité, un Grant, un Chemin, une Exécution et un Enregistrement. Un modèle de menaces commence par lister les principals et pose les cinq questions à chacun. Ledgerly compte neuf types d'acteurs qui méritent d'être listés. Deux d'entre eux, le fournisseur et l'attaquant externe, ne sont pas des principals au sens strict, et c'est précisément pourquoi ils ont leur place sur la liste : il est facile de les en oublier.

Services

Les services (`billing-api`, `invoice-worker`, `payments-service` et `admin-console`, `reporting-job` étant traité ensuite comme un job) sont les plus gros utilisateurs de la base de données. Ils tournent sans surveillance, gardent leurs credentials dans leur configuration et ne se comportent qu'aussi correctement que leur code. Un service peut mal se comporter sans que personne ne fasse rien de mal : une boucle de retries qui ouvre des connexions jusqu'à atteindre la limite, une requête qui parcourt toute une table parce qu'un index a été supprimé, un déploiement qui fait pointer un build de staging vers le primary.

Dans l'état initial, chaque service se connecte avec le même login admin, si bien que le rayon d'impact de chaque service est la base de données entière, alors que `billing-api` et `invoice-worker` n'ont chacun besoin que d'une poignée de tables. Un défaut dans le plus petit service a la portée du plus grand.

Batch jobs

`reporting-job` mérite sa propre ligne parce qu'un job tourne selon une planification, souvent la nuit, sans que personne ne le surveille. Il tend à lire beaucoup de données, à garder de longues transactions ouvertes et à produire des fichiers. Dans l'état initial, il lit parfois le primary, et son étape d'export écrit les fichiers du fournisseur avec le login admin partagé. Son comportement normal ressemble déjà à un accès en masse, ce qui rend son comportement anormal plus difficile à repérer que celui d'une personne.

Ingénieurs d'astreinte

La rotation d'astreinte est le groupe qui subit la plus forte pression pour disposer d'un accès large, et pour de bonnes raisons : un appel d'astreinte en pleine nuit n'est pas le moment d'attendre une approbation que personne n'est éveillé pour donner. Dans l'état initial, ses membres utilisent le mot de passe admin partagé via des clients de bureau qui le mémorisent et font un commit à chaque instruction. La combinaison de l'urgence à corriger et de l'absence de friction pour changer quoi que ce soit en fait le groupe où les accidents sont les plus probables, et le groupe que la conception peut le mieux protéger.

Support

Le personnel du support utilise `admin-console` pour consulter un client à la fois. La menace ici est le périmètre, non la malveillance. Un outil capable d'afficher chaque client, adossé à un rôle capable de lire chaque colonne, répond à une question sur une facture avec un système qui pourrait montrer toutes les factures. La question de conception est de rendre la capacité large indisponible quand seule la capacité étroite est nécessaire.

L'équipe data

L'équipe data travaille sur `ledgerly_analytics`, qui, dans l'état initial, est une restauration nocturne d'un dump non masqué : chaque colonne Restricted et Confidential, à la portée d'un groupe dont le métier est d'examiner des données en masse. Son accès est légitime. Le problème est que la

copie a une audience plus large, une manipulation plus lâche et moins de logs que le primary, si bien qu'un contrôle sur le primary ne protège pas ce qui se trouve dans la copie.

Fournisseurs et intégrations

Le fournisseur comptable externe ne se connecte jamais à une base de données de Ledgerly. Il reçoit des fichiers. Cela change la forme de la menace : l'exposition n'est pas un login mais un export, et son rayon d'impact correspond aux colonnes que contient l'export et à la durée pendant laquelle la copie existe chez le fournisseur. Les questions pertinentes sont ce que contient le fichier, qui en est responsable ensuite, et si quelqu'un remarquerait que le fichier a gagné une colonne.

CI/CD

Le pipeline de déploiement est lui aussi un principal, même s'il relève de l'infrastructure plutôt que d'une personne ou d'un service. Dans l'état initial, il exécute les migrations avec le même login admin, parce que le rôle admin est propriétaire de toutes les tables et que rien d'autre ne peut les modifier, et le credential se trouve dans la configuration du pipeline. Un pipeline qui peut modifier n'importe quelle table et détient un credential permanent représente à la fois un risque de compromission et un risque d'erreur, puisqu'une mauvaise migration s'exécute avec tous les privilèges du propriétaire.

L'équipe plateforme et les DBA

Le petit groupe plateforme gère les bases de données, et c'est le seul acteur qui peut modifier les contrôles eux-mêmes : les rôles, les grants, le point d'entrée par lequel passent les humains, le log. Dans l'état initial, il utilise le login admin partagé comme tout le monde, si bien que rien ne distingue son travail de celui de n'importe qui. Dans la conception cible, il détient un rôle de gestion des accès, `ledgerly_access_admin`, que trois principals utilisent via leurs propres logins. Cela ne supprime pas la menace ; cela la déplace. Un grant erroné ou un paramètre de log désactivé par quelqu'un qui dispose de ce rôle change la signification de chaque autre ligne du modèle.

Pour cette raison, le modèle liste le **plan de contrôle** comme un actif à part entière : l'access gateway, le credential broker, `ledgerly_access_admin` et le stockage d'audit. Les données sont protégées par les contrôles, et les contrôles sont protégés par un ensemble bien plus restreint de personnes et de systèmes. L'exposition est faible en volume et forte en effet, et les questions sont les mêmes que pour toute porte : qui peut l'atteindre, que pourrait-il modifier, et qui le remarquerait ?

L'attaquant externe

Un attaquant externe n'est pas un principal tant qu'il n'a pas obtenu de credential ; la manière utile de le modéliser est donc de partir de ce qu'il lui faudrait obtenir, et non de ce qu'il pourrait faire. Les questions du défenseur sont : quelles portes acceptent un credential susceptible de fuiter, qu'atteindrait-il, et comment nous en apercevrions-nous ? Les réponses viennent des huit autres acteurs. Un credential qui a fuité hérite du rayon d'impact de celui à qui il appartenait.

Trois classes de menaces

Les mêmes acteurs peuvent causer du tort de trois manières différentes, et les contrôles diffèrent pour chacune. Mélanger les classes mène à des contrôles inadaptés : une approbation de l'intention, conçue contre l'usage abusif, ne fait pas grand-chose contre un accident, car l'approbateur convient que la tâche est légitime et l'instruction tourne mal quand même.

Erreurs

Une erreur est une action censée être correcte qui ne l'était pas : une instruction exécutée sur le mauvais environnement, une condition qui correspond à plus de lignes que prévu, une requête trop lourde pour le primary, un script pointé vers la production par une configuration périmée. L'acteur est un principal qui fait son travail.

Je m'attendrais à ce que cette classe soit la plus fréquente dans la plupart des équipes, pour une raison structurelle plutôt que statistique. L'usage abusif exige une personne avec un mobile, et la compromission exige un attaquant avec une occasion, mais une erreur n'exige qu'une personne avec une tâche. Je n'ai pas de chiffre sur l'écart de fréquence des erreurs, et je ne ferais pas confiance à un chiffre si l'on m'en proposait un. La

conclusion pour la conception tient dans les deux cas : des contrôles visant les erreurs protègent contre la classe que toute équipe connaît, qu'elle soit ou non confrontée un jour aux deux autres.

La meilleure façon de traiter les erreurs est de restreindre ce qu'une seule instruction peut atteindre, puisqu'on ne peut pas demander aux gens d'être prudents éternellement. Des rôles qui ne peuvent pas toucher les tables dont ils n'ont pas besoin (Chapitre 3), des timeouts qui arrêtent une requête incontrôlée (Chapitre 5), l'analyse de chaque instruction avant son exécution (Chapitre 9), une voie distincte pour les écritures avec une revue selon le principe des quatre yeux (chapitres 8 et 10) et une manière revue de réparer les données après coup (Chapitre 10) relèvent tous de cette classe.

L'étape des quatre yeux n'aide ici qu'à une condition : le relecteur voit l'instruction et son effet attendu. Approuver l'intention (« corriger la facture 1042 ») est un contrôle pour la classe de l'usage abusif, et il laisse la classe des erreurs intacte. Approuver l'instruction et le nombre de lignes qu'elle devrait modifier traite les erreurs.

Usage abusif et curiosité

L'usage abusif désigne ici un principal disposant d'un accès légitime qui s'en sert pour quelque chose qui sort de la tâche : lire le dossier d'un client par intérêt, exporter davantage que ce dont le travail a besoin, regarder les données de collègues. Il inclut la simple curiosité, qui en est probablement la forme la plus courante et la plus difficile à qualifier de faute. Il inclut aussi le cas délibéré, plus rare.

La réponse de conception n'est pas la suspicion envers l'équipe. Une équipe n'a pas besoin de supposer la mauvaise foi pour remarquer qu'un grant permanent rend la navigation possible et ne laisse aucune trace. Les contrôles qui comptent gardent l'accès lié à une tâche et le rendent visible : des rôles en lecture seule qui retiennent les colonnes Restricted, des vues masquées des données Confidential (Chapitre 12), des grants limités dans le temps (Chapitre 8) et un log d'audit qu'un relecteur lit réellement (Chapitre 14). Sans preuve, on ne peut répondre pour personne à la question « cet accès était-il approprié ? », y compris pour les personnes qui n'ont rien fait de mal.

Compromission

La compromission signifie qu'un credential ou une machine se trouve entre les mains de quelqu'un qui n'est pas le principal prévu : un mot de passe sorti par un fichier de configuration, un ordinateur portable perdu, une variable de pipeline lue par quelque chose à quoi il ne devrait pas faire confiance. Le défenseur n'a pas besoin de savoir comment c'est arrivé pour concevoir en tenant compte du fait que cela peut arriver. La conception contrôle ce que le credential pourrait faire, pendant combien de temps, et si quelqu'un le saurait.

C'est dans la compromission que les propriétés d'un credential comptent le plus. Un credential partagé, statique et sans restriction transforme la perte d'un ordinateur portable en perte de toutes les tables. Un credential limité à un rôle, de courte durée et délivré pour une seule session transforme le même événement en un événement borné et visible. Les chapitres 4 et 7 traitent de la manière dont les credentials sont délivrés ; le Chapitre 3, de ce qu'ils doivent porter, c'est-à-dire le moins possible ; le Chapitre 14, de la manière de s'en apercevoir.

> **Remarque :** Classer un événement relève souvent du jugement. Une correction d'une ligne qui atteint tout un lot est une erreur. La même instruction exécutée volontairement par quelqu'un qui n'a aucune tâche est un usage abusif, et exécutée avec un credential pris sur un ordinateur portable, c'est une compromission. La base de données voit chaque fois la même instruction, ce qui plaide pour des logs qui nomment un login et pour des contrôles qui ne dépendent pas de la devinette des intentions.

Ce qui est protégé

Trois propriétés des données méritent d'être protégées, et le modèle doit dire laquelle chaque menace touche. Ce sont les familières confidentialité, intégrité et disponibilité, et chacune a une signification propre à Ledgerly.

La **confidentialité** porte sur qui peut lire quoi. Chez Ledgerly, le poids est inégal. `iban`, `account_holder_name` et `token_value` sont Restricted, si bien que leur exposition serait difficile à réparer. `contact_name`, `contact_email`, `contact_phone` et `billing_address` sont des PII (don-

nées à caractère personnel identifiantes, personally identifiable information) Confidential des clients. Les montants de `invoices` et de `payments` sont Internal. Une menace qui touche des données Internal mérite moins d'effort de conception qu'une menace qui touche des données Restricted, et le modèle doit montrer cette différence au lieu de traiter toutes les colonnes comme égales. Les régimes de protection des données de ce type tendent à demander quelles données personnelles vous détenez et qui peut les voir, ce qui est la même différence ; l'interprétation relève de vos collègues des services juridique et conformité.

L'**intégrité** porte sur le fait que les données restent correctes. Pour la plupart des tables, une erreur peut être corrigée par une écriture ultérieure, mais `ledger_entries` et `audit_events` sont append-only, et pour le grand livre, l'intégrité compte davantage que le secret : une écriture erronée se corrige par une nouvelle écriture de correction via `payments-service`, jamais par une modification. Une menace d'intégrité sur ces deux tables est donc d'une nature différente. C'est aussi celle où la preuve compte le plus, puisqu'un grand livre qui peut être modifié discrètement est un grand livre sur lequel personne ne peut s'appuyer.

La **disponibilité** porte sur le fait que la base de données serve les services qui en dépendent. C'est la propriété la plus souvent oubliée dans les discussions de sécurité, et celle où les accidents dominent. Un long verrou, une requête qui parcourt une grande table sur le primary, ou un afflux de connexions provenant d'une boucle de retries peuvent arrêter la facturation et l'encaissement des paiements aussi complètement qu'une attaque. La disponibilité obtient des lignes de premier rang dans le modèle, liées à la connection policy (Chapitre 5) et aux endroits où les lectures sont autorisées à aller (Chapitre 6).

Une quatrième propriété se tient à côté de ces trois-là : l'**imputabilité**, c'est-à-dire la capacité de l'enregistrement à dire quel principal a fait quoi. C'est la partie Enregistrement de l'accès. Dans l'état initial, elle est absente, puisque le log ne montre que le rôle partagé, et cette absence augmente le coût de chaque autre menace.

Rayon d'impact

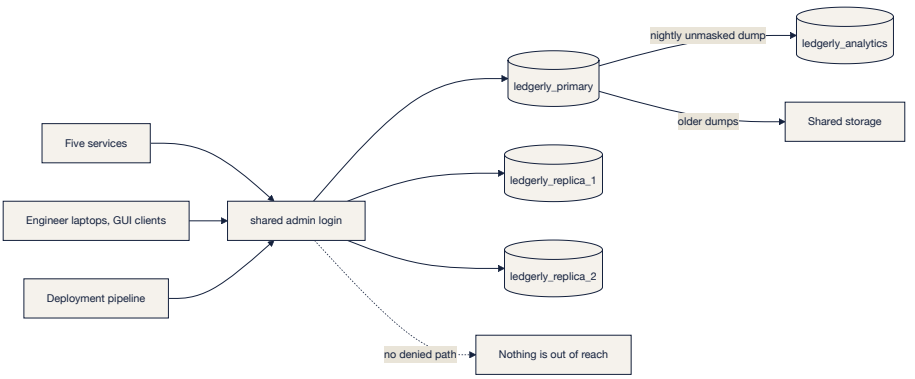
Le modèle a besoin d'un outil de plus pour comparer les menaces : une manière de dire jusqu'où se propage une défaillance unique. Le **rayon d'impact** est l'étendue de ce qui peut être affecté si un principal, un credential ou une action tourne mal. Le terme n'est utile que lorsqu'on l'emploie à propos d'une chose précise ; la question est donc toujours « quel est le rayon d'impact de ce login ? » et jamais « quel est notre rayon d'impact ? »

Un rayon d'impact a sept dimensions, et il aide d'examiner chacune séparément :

- **Portée sur les données :** les tables, colonnes et lignes que le principal peut toucher.
- **Portée sur les actions :** s'il peut lire, écrire, supprimer ou modifier le schéma.
- **Portée sur les environnements :** les bases de données auxquelles il peut se connecter : le primary, les répliques, la copie d'analytics.
- **Portée dans le temps :** combien de temps l'accès reste valide, d'une session à pour toujours.
- **Portée sur les copies :** où d'autre les données qu'il peut lire ont été dupliquées, et qui peut atteindre ces copies.
- **Portée de l'instruction :** ce qu'une seule instruction peut modifier avant que quoi que ce soit ne l'arrête : les lignes qu'elle touche, si elle s'exécute dans une transaction qui peut faire l'objet d'un rollback, et si un timeout la borne.
- **Origine réseau :** quels réseaux et quelles machines peuvent présenter le credential tout court.

Le login admin partagé de l'état initial de Ledgerly atteint le maximum sur chaque dimension. Il atteint toutes les tables et toutes les colonnes, il peut lire, écrire, supprimer et modifier, il se connecte au primary et aux deux répliques, il n'a pas d'expiration, et ses données ont déjà des copies en dehors du primary. Toute instruction est illimitée et fait un commit immédiatement, et n'importe quel ordinateur portable sur le VPN peut présenter le

credential. Le diagramme montre ce qu'il faut regarder : tout ce qui est à gauche détient le même credential, donc tout ce qui est à droite se trouve dans un seul rayon.



Comparez-le avec le rôle de l'état cible `app_reporting_job` : en lecture seule, sur une réplique, sans accès aux colonnes Restricted. Un défaut qui pourrait exposer des données Restricted dans l'état initial se réduit, pour ce rôle, à une requête lourde sur une réplique. Le défaut est identique ; le rayon d'impact a changé.

Pour que la comparaison reste honnête, écrivez le rayon login par login, sous forme de liste de questions et non de score, et placez deux logins côte à côte. L'esquisse ci-dessous est illustrative : une aide à la revue, et non un outil sur lequel le livre s'appuie. Une réponse inconnue est enregistrée comme inconnue, car un modèle qui arrondit les inconnues à « ça va » cache les cas pour lesquels il existe.

```
dimension          shared admin login          app_reporting_job
data reach         every table                 unknown until Chapter 3
lists the tables
restricted cols    all                         none
action reach       read, write, delete, DDL    read
environment        primary, both replicas      one replica
time reach         no expiry                   unknown until Chapter 4
decides
copy reach         unmasked restore, dumps     the export files it
writes
statement reach    unbounded, commits at once  read-only transaction,
statement timeout
network origin     company VPN, any laptop     unknown until Chapter 7
decides
```

La comparaison est utile parce qu'elle est inégale. Là où la seconde colonne dit « unknown », le modèle a trouvé une question à laquelle la conception n'a pas encore répondu, et c'est un constat, non un trou dans le tableau.

Deux limites s'appliquent. Un rayon décrit ce qu'un login pourrait faire, non ce qu'il fera ; c'est donc une limite de planification et non une prévision. Et il ne dit rien de la probabilité de chaque menace. Un rayon étroit sur une porte ouverte toute la journée peut quand même mériter plus d'attention qu'un rayon large sur une porte qui s'ouvre une fois par trimestre, et c'est pourquoi le modèle doit combiner le rayon avec l'exposition.

Le modèle de menaces de Ledgerly

Une fois les acteurs, les classes, les propriétés et le rayon d'impact en place, le modèle tient dans un seul tableau. Le tableau ci-dessous couvre l'état initial de Ledgerly. Chaque ligne correspond à un acteur et un chemin, avec la classe de menace qui compte le plus pour cette paire, l'impact sur les propriétés ci-dessus, les chapitres qui fournissent le contrôle, et le **risque résiduel** : ce qui reste vrai une fois ces contrôles en place. L'« exposition » est un jugement qualitatif (élevée, moyenne ou faible) de la fréquence d'utilisation de la porte et de la facilité à en faire un usage abusif, et elle est écrite pour que quelqu'un puisse la contester.

Acteur	Chemin	Classe principale	Impact	Exposition	Contrôlé dans	Risque résiduel
Services (les cinq)	Login admin partagé tiré de la configuration de chaque service	Erreur, compromission	N'importe quelle table, toutes les colonnes ; disponibilité si les connexions s'épuisent	Élevée	Chapitres 3, 4, 5	Un défaut garde la portée de son propre rôle
`reporting-job`	Lit le primary lors de certaines exécutions	Erreur	Les lectures lourdes concurrencent les écritures ; disponibilité du primary	Moyenne	Chapitres 5, 6	Une lecture lourde sur une réplique ralentit encore les autres lecteurs de la réplique

Acteur	Chemin	Classe principale	Impact	Exposition	Contrôlé dans	Risque résiduel
Ingénieurs d'astreinte	Ordinateur portable, client graphique, mot de passe admin partagé, autocommit	Erreur, compromission	Écritures sans périmètre ; intégrité de n'importe quelle table	Élevée	Chapitres 7, 8, 10	Une instruction saisie à la main dans un rôle accordé peut encore être erronée
Support	`admin-console` agissant sous le login partagé	Usage abusif, erreur	Navigation dans les données client Confidential ; modifications sur le mauvais client	Moyenne	Chapitres 3, 10, 12	Le support voit encore les colonnes dont le dossier d'un client a besoin
Équipe data	Restauration d'analytics non masquée et anciens dumps	Usage abusif, compromission	Confidentialité de chaque colonne Restricted et Confidential	Élevée	Chapitres 12, 13	Des données masquées peuvent encore révéler des valeurs par inférence
Fournisseur comptable	Fichiers exportés	Erreur, usage abusif	Confidentialité des colonnes exportées et leur durée de vie	Moyenne	Chapitres 12, 13	Le fournisseur garde le fichier après la livraison
Pipeline de déploiement	Login admin stocké comme variable de pipeline	Erreur, compromission	Intégrité du schéma et des données ; toutes les tables	Moyenne	Chapitres 4, 11	Une migration revue s'exécute encore avec les droits du propriétaire
L'équipe plateforme et les DBA	Login admin partagé aujourd'hui ; `ledgerly_access_admin` dans la conception cible	Erreur, usage abusif, compromission	Les grants, les contrôles et les preuves peuvent être modifiés	Moyenne	Chapitres 3, 7, 14	Le rôle existe, donc son usage abusif est possible ; seule la détection reste

Acteur	Chemin	Classe principale	Impact	Exposition	Contrôlé dans	Risque résiduel
Plan de contrôle (actif)	Gateway, credential broker, stockage d'audit	Erreur, compromission	Chaque autre ligne change de sens	Moyenne	Chapitres 4, 7, 14	La compromission d'un composant affaiblit les preuves qu'il détient
Toute session humaine	Une instruction saisie à la main dans un rôle accordé	Erreur	Instruction illimitée, commit immédiat	Élevée	Chapitres 9, 10	Une instruction peut être valide, approuvée et pourtant erronée dans son effet
Toute origine réseau	Le VPN de l'entreprise atteint directement le primary	Compromission	Un credential qui a fuité fonctionne depuis n'importe quel ordinateur portable sur le VPN	Moyenne	Chapitre 7	Un credential volé depuis une origine autorisée y fonctionne encore
Attaquant externe	Tout credential ayant fuité depuis les lignes ci-dessus	Compromission	Hérite du rayon du propriétaire du credential	Faible à moyenne	Chapitres 4, 7, 14	Un credential volé pendant sa durée de vie fonctionne jusqu'à son expiration
Chaque acteur	Aucun login nominatif dans le log de la base de données	Toutes les classes	Imputabilité de chaque autre ligne	Élevée	Chapitres 7, 14	L'enregistrement n'est digne de confiance qu'autant que le stockage qui le détient

Quatre observations découlent du tableau, et ce sont elles qui justifient de le construire.

Premièrement, le login admin partagé apparaît dans la plupart des lignes. Le tableau montre un seul problème aux multiples visages, et le premier contrôle, un rôle distinct par principal (Chapitre 3), en traite une grande

partie d'un coup. Un modèle qui révèle l'ordre dans lequel travailler fait son travail.

Deuxièmement, parmi les expositions les plus élevées figurent des portes qui n'ont jamais été censées en être : la restauration d'analytics et les anciens dumps. Aucune n'est un chemin d'accès dans le schéma mental que quiconque se fait de la base de données. C'est pourquoi les chapitres 6 et 13 traitent chaque copie de données comme une porte avec un responsable et une expiration.

Troisièmement, la colonne du risque résiduel est la partie honnête du tableau. Chaque contrôle laisse quelque chose : un rôle accordé peut encore faire l'objet d'un usage abusif, une instruction valide peut encore être erronée, un enregistrement n'est digne de confiance que dans la mesure où son stockage l'est. Une ligne sans risque résiduel écrit est une ligne où personne n'a posé la question.

Quatrièmement, les lignes du plan de contrôle, de la session humaine et de l'origine réseau nomment un actif ou un chemin plutôt qu'un acteur, et la dernière ligne ne nomme pas non plus un acteur unique. L'absence d'imputabilité est une propriété du système entier, et elle augmente le coût de chaque autre ligne, parce que sans login nominatif aucun incident ne peut être circonscrit. Le Chapitre 14 ferme cette ligne. Les chapitres précédents construisent les identités qui rendent sa fermeture possible.

> **Avertissement :** Un modèle de menaces complet est une carte des points où Ledgerly est le plus faible. Conservez-le dans un endroit soumis aux mêmes restrictions d'accès que les systèmes qu'il décrit, et ne le collez pas dans un ticket public ou dans un wiki à large audience. Un document qui liste des copies non masquées et des credentials partagés est utile aux personnes qui le maintiennent et à personne d'autre.

Garder le modèle vivant

Un modèle de menaces écrit une seule fois est un instantané, et le tableau ci-dessus est périmé le jour où Ledgerly ajoute une intégration ou où l'équipe data obtient une nouvelle copie. Je le traiterais comme un document vivant avec un responsable nommé, comme un runbook, conservé

dans un dépôt soumis aux mêmes restrictions de lecture que les systèmes qu'il décrit, puisque le tableau liste des faiblesses. Cela a un coût. La configuration des accès vit généralement dans un dépôt plus largement lisible, si bien qu'un changement à cet endroit n'apparaît pas automatiquement dans la revue du modèle. J'accepterais ce coût et je comblerais l'écart par une règle : une pull request qui modifie la configuration des accès renvoie vers la modification correspondante du modèle, ou explique pourquoi aucune n'est nécessaire.

Certains événements doivent déclencher une mise à jour sans que personne n'ait à s'en souvenir :

- Un nouveau service, job, fournisseur ou outil qui atteint les données de production.
- Une nouvelle copie de données : une nouvelle réplique, un export, un rafraîchissement de staging, un dump pour une analyse ponctuelle.
- Un changement dans qui peut atteindre quoi : une réorganisation, une nouvelle rotation d'astreinte, un contrat fournisseur qui prend fin.
- Un incident ou un quasi-incident, qui est une preuve de la ligne qui était fausse.
- Un constat lors d'une revue des accès (Chapitre 14), qui est une preuve de l'hypothèse qui a dérivé.

Chaque ligne doit enregistrer assez d'éléments pour qu'un inconnu puisse la contester : l'acteur, le chemin, la classe, l'impact, le jugement d'exposition, le contrôle et son responsable, la date de la dernière revue, et l'hypothèse dont dépend la ligne. Le dernier champ est celui que les équipes sautent, et le plus utile : « l'export fournisseur ne contient que ces colonnes » est une affirmation vérifiable, et une ligne qui dépend d'une affirmation non vérifiée est une supposition.

Une petite vérification peut trouver les lignes devenues périmées. Elle ne décide de rien. Elle oriente un humain vers les lignes les plus susceptibles d'être fausses, et elle traite une date de revue manquante comme périmée. Elle prend aussi en entrée les événements listés ci-dessus : quand un pipeline crée un login, un rôle ou une copie de données, il enregistre un événement, et un événement qu'aucune ligne ne couvre ouvre un élément de revue, si bien que la mise à jour ne dépend de la mémoire de personne.

```
function findStaleThreatRows(rows, maxAgeDays, accessEvents):
    stale = []
    // an event from the pipeline: a new login, role or data copy
    for each event in accessEvents:
        if not anyRowCovers(rows, event):
            stale.append(newReviewItem(event))
    for each row in rows:
        // a row with no review date has never been reviewed
        if row.last_reviewed is null:
            stale.append(row)
        else if now() - row.last_reviewed > maxAgeDays:
            stale.append(row)
        // a control with no owner is a control nobody maintains
        else if row.control_owner is null:
            stale.append(row)
    if count(stale) > 0:
        notify("platform", "threat model rows need review")
    return stale
```

La cadence est un compromis. Une passe trimestrielle est quelque chose qu'une petite équipe peut tenir, et une passe annuelle suffit rarement, mais je choisirais l'intervalle selon la vitesse à laquelle les portes de Ledgerly changent. Les contacts sécurité et conformité fixent les questions que pose la revue, et l'équipe plateforme y répond avec des faits sur le système.

Compromis et coûts

Un modèle de menaces coûte du temps à écrire et davantage de temps à entretenir. Pour une petite équipe, la première version prend vraisemblablement quelques jours de discussion attentive, et la maintenance est un élément récurrent qui entre en concurrence avec le travail sur les fonctionnalités. Si personne n'en est responsable, le coût est payé une fois et le bénéfice s'érode. Dans un très petit système avec un service et une base de données, le tableau complet est plus que ce dont le problème a besoin ; une demi-page listant les portes et le rayon du pire cas de chacune suffit.

Le modèle peut aussi créer une fausse confiance : un tableau bien rangé suggère que chaque menace y figure. Il liste les menaces auxquelles quelqu'un a pensé, et une porte non listée lui est invisible. La meilleure défense, ce sont les déclencheurs de mise à jour ci-dessus, et l'habitude de demander après chaque incident : « quelle ligne aurait prédit cela, et si aucune, quelle est la ligne manquante ? »

Les jugements d'exposition sont subjectifs. Deux ingénieurs évalueront la même porte différemment, et le désaccord est instructif, mais une note suggère plus de précision que ce que les preuves permettent. Je m'en tiendrais à trois niveaux et je résisterais à la tentation de calculer des scores.

Lister des groupes de personnes comme acteurs a un coût humain. Qualifier les ingénieurs d'astreinte ou le support d'acteurs de menace peut sonner comme une accusation. La réponse tient dans le cadrage : le modèle décrit des chemins et des rôles, non des individus, et ses entrées disent ce que la conception permet. Les équipes du tableau y figurent parce qu'elles ont le besoin le plus légitime d'atteindre les données.

Quand ne pas utiliser le modèle complet : si personne ne peut actuellement répondre à « qui a accès ? », commencez par un inventaire des logins et des copies de données avant d'écrire des menaces, car un modèle construit sur un inventaire inexact décrit un système qui n'existe pas. Et ne l'utilisez pas comme substitut à une décision. Il classe et explique ; c'est toujours l'équipe qui choisit quels contrôles construire en premier et quels risques accepter par écrit.

Erreurs courantes

- **Ne modéliser que l'attaquant externe.** Cela correspond à l'image courante de la sécurité, et les contrôles sont familiers. Le coût, c'est que les erreurs et la curiosité ne sont pas traitées. Donnez à chacune des trois classes ses propres lignes dans le tableau.
- **Lister des contrôles au lieu de menaces.** « Nous avons un VPN » enregistre une réponse sans la question. Énoncez d'abord ce qui peut mal tourner, puis le contrôle, afin qu'un contrôle puisse être retiré quand sa menace ne s'applique plus.
- **Omettre les acteurs qui ne sont pas des personnes.** Les pipelines, les jobs et les exports sont faciles à négliger parce que personne ne se connecte sous leur identité, et ce sont eux qui tendent à détenir le plus de pouvoir permanent. Listez chaque principal et chaque export.
- **Traiter les données comme uniformément sensibles.** Si chaque colonne est « critique », le modèle ne donne aucune indication sur où commencer. Utilisez les quatre classes de données (Restricted, Confi-

dential, Internal, Public) afin que le poids d'une menace suive le poids des données.

- **Négliger la disponibilité.** Une requête incontrôlée ne ressemble pas à un événement de sécurité, si bien qu'elle échappe aux revues de sécurité. Pour une activité de paiements, elle peut compter tout autant. Donnez à la disponibilité ses propres lignes.
- **L'écrire une seule fois.** Le modèle est exact pendant quelques semaines, puis devient du folklore. Désignez un responsable et ajoutez des déclencheurs de mise à jour.

Points clés

- Un modèle de menaces est une courte liste structurée d'acteurs, de chemins, de classes de menaces, d'impacts et de contrôles, et son rôle est de permettre à une équipe de comparer les portes entre elles.
- Les erreurs, l'usage abusif et la compromission exigent des contrôles différents, et une conception qui ne considère que les personnes extérieures laisse les deux premières classes sans réponse ; approuver l'intention traite l'usage abusif, tandis qu'approuver l'instruction et son effet attendu traite les erreurs.
- Chaque acteur a sa place sur la liste, y compris les pipelines, les batch jobs, les exports vers les fournisseurs et l'équipe plateforme qui gère les contrôles, parce que les principals sous lesquels personne ne se connecte détiennent souvent le plus de pouvoir permanent, et le plan de contrôle est lui aussi un actif.
- La confidentialité, l'intégrité et la disponibilité ont chacune besoin de leurs propres lignes, et l'imputabilité, la capacité de dire quel principal a fait quoi, augmente le coût de chaque autre menace lorsqu'elle manque.
- Le rayon d'impact porte toujours sur un login, un credential ou une action précis, et il se décrit selon sept dimensions : données, actions, environnement, temps, copies, instruction et origine réseau.
- Dans l'état initial de Ledgerly, un seul login admin partagé apparaît dans presque chaque ligne ; un rôle distinct pour chaque principal est donc le contrôle qui réduit d'un coup la plus grande part du tableau.

- Le modèle ne vaut que par son entretien : donnez-lui un responsable, liez les mises à jour à des événements tels que de nouveaux services et de nouvelles copies de données, et conservez-le là où seules les personnes qui maintiennent le système peuvent le lire.

Le modèle vous dit quelles portes comptent le plus, et dans le cas de Ledgerly, presque toutes ramènent à un seul login. Le Chapitre 3 commence à démonter ce login : qui sont les principals, ce qu'un rôle doit contenir, et à quel point chacun d'eux a besoin de peu.

Chapitre 3 — Identités, rôles et moindre privilège

Imaginez un jeudi soir chez Ledgerly. Une nouvelle table, `invoice_disputes`, a été mise en service mardi, et le contact sécurité pose à l'équipe plateforme une question courte et raisonnable : qui peut la lire ? Alex commence par les grants et n'en trouve aucun, parce que chaque appelant se connecte en tant que `admin`, qui est propriétaire de la table et peut tout lire. Alex cherche ensuite qui l'a réellement lue. Le log de la base de données contient des milliers de lignes pour cette table, toutes écrites par le même login : le worker, le rapport nocturne, deux ingénieurs sur leurs ordinateurs portables et l'outil de support.

Personne ne peut situer la nouvelle table. La réponse honnête à « qui peut la lire ? » est « quiconque peut atteindre la base de données », et la réponse honnête à « qui l'a lue ? » est « quelqu'un ». Personne n'a rien fait de mal. La conception n'a jamais demandé à la base de données de distinguer ces appelants, ni de limiter aucun d'entre eux.

Le Chapitre 1 a divisé l'accès en cinq parties : **Identité**, **Grant**, **Chemin**, **Exécution** et **Enregistrement**. La thèse du livre est que l'accès est un problème d'autorisation, d'exécution et d'imputabilité, et non un problème de distribution de credentials. Ce chapitre sert le pilier de l'autorisation. Il couvre les deux premières parties, qui demande (Identité) et ce que ce principal peut faire (Grant), et il marque l'endroit où les grants s'arrêtent, parce que la partie Exécution prend le relais à cet endroit (Chapitre 9).

Il ajoute les premières étapes de l'état cible de l'exemple qui court tout au long du livre. Chaque table appartient à `ledgerly_owner`, et personne ne se connecte en tant que propriétaire. Chaque service reçoit son propre rôle applicatif avec les grants les plus étroits qui fonctionnent. Le chapitre présente aussi le schéma de rôles complet, la manière dont les principals at-

teignent les rôles, les grants au niveau des colonnes et le plafond de l'accès just-in-time. Les chapitres suivants s'appuient sur ces définitions et ne les répètent pas.

Le problème, et pourquoi il est difficile

Une base de données répond à deux questions pour chaque instruction : qui est-ce, et ce principal a-t-il le droit de faire cela ? Un compte partagé vide la première réponse de son sens, et un compte large rend la seconde réponse toujours positive. On arrive facilement aux deux. Un login, un mot de passe et tous les droits, c'est la chose la moins coûteuse qui fonctionne le jour où une équipe crée sa première base de données.

Le coût arrive plus tard, sous forme de preuves manquantes. Quand les personnes et les services qui partagent le login admin (disons vingt personnes, cinq services, le pipeline de déploiement et l'étape d'export) l'utilisent tous, le log de la base de données ne peut pas dire lequel d'entre eux a exécuté une instruction. Le log applicatif décrit ce que l'application voulait faire, non ce qui a atteint la base de données. C'est un problème de conception, non un problème de discipline, car aucune prudence individuelle ne ramène une information que la conception a jetée.

Le remède évident est d'écrire une politique : n'accorder que ce qui est nécessaire. Une politique ne fonctionne pas ici, parce que la base de données fait du choix large le choix facile. Tout accorder prend une instruction et n'échoue jamais. Accorder l'ensemble le plus étroit qui fonctionne exige de connaître chaque requête qu'exécute un workload, y compris le job mensuel dont personne ne se souvient et le chemin de retry qui s'exécute deux fois par an. Un grant trop étroit casse la production au pire moment. Un grant trop large ne casse rien, si bien que personne ne le remarque. Avec cette asymétrie, les privilèges ne font que croître.

La propriété ajoute un second piège. Dans la plupart des moteurs, le propriétaire d'une table peut en modifier la structure, la supprimer et en accorder l'accès à d'autres. Si un service se connecte en tant que propriétaire parce qu'il crée des tables au démarrage, le service détient le pouvoir de réécrire le modèle d'accès de la base de données. Si une personne est pro-

priétaire, le modèle dépend de l'emploi de cette personne. La propriété est un privilège structurel, et elle ne devrait résider ni dans quelque chose qui traite du trafic ni dans quelque chose qui a une carrière.

Le remède est une structure, non une consigne : un login distinct par principal, un petit ensemble de rôles qui portent les privilèges, et un moyen de garder les deux en phase avec ce que l'équipe entend.

Principals, logins et rôles

Ce chapitre travaille sur l'Identité et le Grant, et touche le Chemin là où un rôle dit d'où il peut se connecter. Un **principal** est tout ce qui s'authentifie. Dans le monde de Ledgerly, il en existe quatre types qu'il vaut la peine de distinguer.

- **Les humains.** Les ingénieurs backend d'astreinte, le personnel du support, l'équipe data et l'équipe plateforme. Ils arrivent et partent, changent d'équipe et travaillent depuis des endroits différents.
- **Les services.** `billing-api`, `invoice-worker` et `payments-service`. Ils tournent longtemps, s'exécutent en de nombreuses instances et portent le trafic le plus lourd.
- **Les jobs.** `reporting-job` et son étape d'export. Ils s'exécutent selon une planification et se terminent. Un job a généralement besoin de moins qu'un service, et pour moins longtemps.
- **Les outils.** `admin-console` est le cas le plus net. C'est un service par sa manière de se connecter, mais il agit pour une personne, si bien que deux principals interviennent dans chaque action : l'outil et l'humain derrière lui.

Le fournisseur comptable externe ne se connecte jamais à aucune base de données de Ledgerly ; il reçoit des fichiers, et le Chapitre 13 traite ce chemin.

Identité, login et rôle sont trois choses différentes

Le livre tient ces trois mots séparés, parce que les confondre est la source de la plupart des confusions en conception des accès.

Une **identité** est l'affirmation vérifiée qu'un principal est bien celui qu'il prétend être. Un **login** est le compte au niveau de la base de données avec lequel un principal se connecte. Un **rôle** est un ensemble nommé de privilèges. Un principal a une identité, se connecte par un login, et le login est membre d'un ou plusieurs rôles. C'est le login qui apparaît dans le log de la base de données. C'est le rôle qui décide de ce que le login peut faire.

Cela donne au modèle une phrase à retenir : personne ne se connecte en tant que rôle. Un rôle existe pour être accordé à des logins et pour porter des privilèges, et ce n'est jamais une personne. Si le log montre un nom de rôle là où devrait figurer une personne ou une instance de workload, le modèle a été contourné quelque part.

Pour les humains, l'access gateway (Chapitre 7) crée un login par session qui porte l'identité de la personne, et le login reçoit `human_readonly` par défaut. Pour les services, chaque instance se connecte avec un login membre de son rôle `app_*`. Dans l'état cible, ce login est dynamique et de courte durée (Chapitre 4). Dans l'état initial, c'est le login admin partagé. La conséquence pour la question centrale est simple : le log nomme un login, et un login correspond à une personne ou à une instance de workload, si bien que « qui a exécuté cette instruction ? » a une réponse.

Pourquoi un compte partagé détruit la preuve

Le Chapitre 1 a présenté l'argument en entier ; en bref, un compte partagé ne peut pas imputer une instruction, ne peut pas être révoqué pour une seule personne, et ne peut pas être restreint en deçà de ce dont son appelant le plus large a besoin.

Un chemin demande du soin : `app_admin_console` est un workload, mais ses utilisateurs sont des personnes. La console transmet l'identité du principal humain avec chaque action, et l'esquisse ci-dessous montre comment.

```
function consoleAction(person, action):
    correlationId = newCorrelationId()
    outcome = "failed"
    // set on the session the console borrowed, cleared when it is
returned
    setSessionContext({ actor: person.id, correlation: correlationId
})
    try:
        run(action)
        outcome = "done"
    finally:
        clearSessionContext()
        // a failed or half-finished action is recorded too
        auditLogAppend({ actor: person.id, correlation: correlationId,
                         workload: "admin-console", action:
action.name,
                         result: outcome })
```

Le login de la base de données nomme le workload, et le contexte de session et `audit_events` portent la personne et le correlation id, si bien que le log applicatif, la piste d'audit et l'enregistrement propre à l'instruction partagent une même clé. Le Chapitre 14 explique comment les enregistrements sont joints. Le contexte est défini par la console ; c'est donc une affirmation du workload et non un fait vérifié par la base de données ; la réserve sur le pooler de la section sur les règles de ligne s'y applique aussi.

Une hiérarchie de rôles en quatre niveaux

Je construirais le schéma de rôles en quatre niveaux, de celui que personne n'utilise vers ceux que les gens utilisent. Chaque niveau a une seule fonction.

Niveau un: le propriétaire

`ledgerly_owner` est propriétaire de chaque objet du schéma : schémas, tables, vues et séquences. Personne ne se connecte avec lui. Il existe pour que la propriété ne soit rattachée ni à une personne ni à un service.

```
CREATE ROLE ledgerly_owner;
// no login is ever created for this role; objects are created
// under it only by a migration that assumes it

CREATE ROLE ledgerly_migrator;
GRANT ledgerly_owner TO ledgerly_migrator;
// membership grants the owner's privileges, not ownership: objects
// belong to the active role, so the migration always assumes
// ledgerly_owner explicitly, on every engine
```

Niveau deux: le migrator

`ledgerly_migrator` exécute les changements de schéma. Le pipeline de déploiement utilise son login, les humains non, et il ne détient aucun privilège sur les données en propre. Il est membre de `ledgerly_owner`. Sur certains moteurs, cette appartenance transmet les privilèges du propriétaire sans aucune étape explicite, mais pas la propriété : un nouvel objet appartient au rôle actif au moment de sa création, si bien que la migration endosse toujours `ledgerly_owner` explicitement.

Le migrator détient des privilèges de propriétaire permanents, exercés au moyen d'un credential de courte durée. Les privilèges n'expirent jamais ; seul chaque credential expire. Trois choses le bornent : le credential ne vit que dans le pipeline, il est de courte durée (chapitres 4 et 11), et la procédure du pipeline est la seule manière admise de l'utiliser. Les changements de schéma sont possibles sans qu'aucune personne ni aucun service ne détienne la propriété, et si ces bornes tiennent, chaque usage du propriétaire correspond à une exécution du pipeline. Le Chapitre 11 explique comment les migrations sont revues.

Les backfills qui font partie d'un changement de schéma s'exécutent dans la migration en tant que `ledgerly_migrator`. Toute autre modification en masse des données est une correction de données et suit le Chapitre 10. Je trace cette ligne parce que le pouvoir du migrator est grand, et qu'il ne doit pas devenir la voie de chaque changement délicat.

Niveau trois: les rôles applicatifs

Chaque workload reçoit son propre rôle. Un rôle par service coûte cinq définitions à garder en phase avec cinq bases de code. En échange, le rayon d'impact de chaque credential a la taille de la tâche de ce service : si celui

de `invoice-worker` fuit, il expose ce que le worker pouvait faire, et le worker n'a jamais eu besoin de `payment_tokens`.

Les rôles sont `app_billing_api`, `app_invoice_worker`, `app_payments_service`, `app_reporting_job` et `app_admin_console`. Le préfixe dit qui l'utilise : `app_` pour les workloads, `human_` pour les personnes. Les rôles ne portent ni nom de personne ni nom d'environnement.

Niveau quatre: les rôles humains

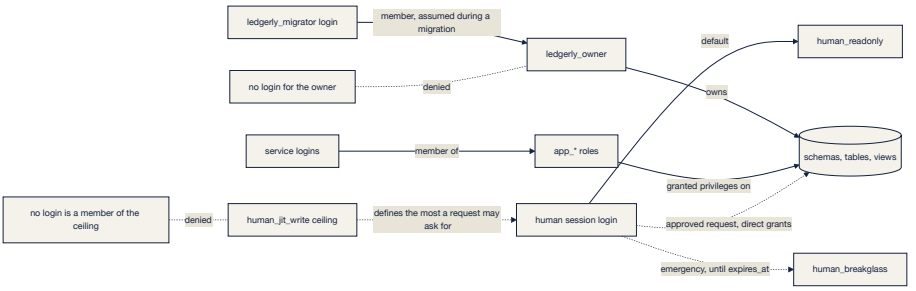
Les personnes reçoivent trois rôles, qui diffèrent par la manière dont ils sont détenus.

- `human_readonly` est le rôle par défaut. Il lit pour enquêter, retient les colonnes Restricted et masque les colonnes Confidential là où le Chapitre 12 le prévoit.
- `human_jit_write` est un plafond. Il n'est jamais accordé à un login. Chaque demande approuvée en accorde un sous-ensemble, sur des tables et des colonnes nommées, directement au login de session, puis le retire.
- `human_breakglass` est un accès d'urgence large avec une journalisation renforcée (le Chapitre 8 le définit) et une revue obligatoire après coup.

Le plafond est le moins intuitif des trois, et j'y reviens après la mécanique des grants.

Le diagramme

Le diagramme ci-dessous montre qui est membre de quoi, et qui est propriétaire de quoi. Regardez les deux éléments qui ne sont pas reliés : aucun login n'existe pour `ledgerly_owner`, et aucun login n'est membre de `human_jit_write`.



Grants: schéma, table et colonne

Un **grant** est l'acte, et l'enregistrement qui en résulte, de donner un privilège à un rôle, de donner un rôle à un login ou à un autre rôle, ou, pour l'accès JIT uniquement, de donner un privilège directement à un login de session. Un **privilège** est la chose précise qu'un grant autorise, comme `SELECT` sur une table. Les trois sens gardent le vocabulaire précis, et dans ce chapitre je dirai lequel j'entends.

Partir de rien

Le point de départ de tout rôle est l'absence totale de privilèges. Ajoutez des grants jusqu'à ce que le workload fonctionne, et arrêtez-vous. Les moteurs diffèrent dans ce qu'un nouveau login peut faire par défaut : sur certains, n'importe quel login peut voir les noms des objets ou créer des objets dans un schéma par défaut. Listez ce que chaque login reçoit automatiquement, et retirez ce dont un rôle n'a pas besoin.

Les privilèges s'appliquent à trois niveaux, et ils s'additionnent : un rôle détient l'union de tout ce qui lui a été accordé à n'importe quel niveau, et un grant étroit ne retire jamais un grant large.

- **Niveau du schéma.** Si un rôle peut voir les objets d'un schéma tout court. Sur les moteurs qui l'exigent, les grants au niveau des tables ne servent à rien sans lui, ce qui fait de l'accès au schéma un interrupteur grossier utile.
- **Niveau de la table.** `SELECT`, `INSERT`, `UPDATE` et `DELETE` sur une table entière.
- **Niveau de la colonne.** Les mêmes privilèges sur des colonnes nommées seulement.

Les grants au niveau des colonnes comptent parce que les données sensibles sont inégalement réparties : `customers` contient des coordonnées Confidential, `bank_accounts` contient `iban` et `account_holder_name`, qui sont Restricted, et `payment_tokens` contient `token_value`, qui est Restricted. Un grant au niveau de la table dit oui à chaque colonne de la table. Comme les privilèges s'additionnent, restreindre des colonnes signifie n'accorder qu'au niveau des colonnes et ne détenir, pour ce rôle, aucun grant au niveau de la table sur cette table, ni directement ni par un autre rôle dont il est membre.

Grants applicatifs

Les deux premiers rôles applicatifs, au niveau des tables :

```
GRANT SELECT, INSERT, UPDATE ON customers, invoices, invoice_lines TO
app_billing_api;
GRANT SELECT ON payments, ledger_entries TO app_billing_api;
GRANT INSERT ON audit_events TO app_billing_api;

GRANT SELECT, INSERT, UPDATE ON invoices, invoice_lines TO
app_invoice_worker;
GRANT SELECT ON customers TO app_invoice_worker;
GRANT INSERT ON audit_events TO app_invoice_worker;

// not granted to either role: bank_accounts, payment_tokens,
// UPDATE or DELETE on ledger_entries and audit_events
```

`app_billing_api` lit `ledger_entries` et `payments` mais ne peut pas y écrire, parce que seul `payments-service` écrit dans `payments`, `bank_accounts` et `payment_tokens` et insère dans `ledger_entries`. Aucun des deux rôles ne détient `DELETE` sur quoi que ce soit ; si un service en a besoin, le grant est ajouté pour cette table avec une justification, et la revue le remarque.

Grants au niveau des colonnes pour les autres rôles

Les rôles marqués d'une croix dans la matrice sont restreints au niveau des colonnes. Les listes de colonnes ci-dessous sont illustratives : quelles colonnes Confidential apparaissent, et sous quelle forme masquée, relève de la décision du Chapitre 12.

```
// illustrative column lists
GRANT SELECT (id, created_at, status) ON customers TO
app_reporting_job;
GRANT SELECT (id, customer_id, amount_minor, currency, status)
    ON payments TO app_reporting_job;
GRANT SELECT ON invoices, ledger_entries TO app_reporting_job;

GRANT SELECT (id, created_at, status) ON customers TO
app_admin_console;
GRANT SELECT, UPDATE (status) ON invoices TO app_admin_console;
// UPDATE (status) is a column-level grant: the tool can change an
invoice's
// status and no other column of the row
```

Le dernier grant est celui que je préfère. `app_admin_console` peut modifier le `status` d'une facture et rien d'autre dans la ligne ; le grant énonce donc l'intention, et la base de données l'impose même si le code de l'outil change plus tard. `app_reporting_job` ne détient aucun privilège d'écriture sur aucune table principale : il écrit ses résultats dans `ledgerly_analytics`.

> **Remarque :** Sur les moteurs qui répliquent physiquement, les rôles sont définis une seule fois pour tout le cluster ; l'endroit d'où chaque rôle peut se connecter est donc une règle de la couche de connexion (chapitres 5 et 7) et non quelque chose que les grants expriment. Le Chapitre 6 donne le détail.

La matrice comme référence

Le tableau ci-dessous est la référence de l'état cible pour les six rôles qui lisent ou écrivent les tables principales. S est `SELECT`, I est `INSERT`, U est `UPDATE`, D est `DELETE`, un tiret signifie aucun accès, et une croix (†) signifie restreint au niveau des colonnes : les colonnes Restricted sont retenues, et les colonnes Confidential sont masquées là où le Chapitre 12 le prévoit.

Table	`app_billing_api`	`app_invoice_worker`	`app_payments_service`	`app_reporting_job`	`app_admin_console`	`human_readonly`
`customers`	S, I, U	S	S	S†	S†	S†
`bank_accounts`	—	—	S, I, U	—	—	—
`payment_tokens`	—	—	S, I, U, D	—	—	S†

Table	app_billing_api	app_invoice_worker	app_payments_service	app_reporting_job	app_admin_console	human_readonly
payments	S	—	S, I, U	S†	S†	S†
invoices	S, I, U	S, I, U	S	S	S, U (status seulement)	S
invoice_lines	S, I, U	S, I, U	—	S†	S†	S†
ledger_entries	S	—	S, I	S	—	S
audit_events	I	I	I	—	I	S

Je lis la matrice par colonne. Chaque colonne est la description de poste d'un workload écrite sous forme de privilèges, et un relecteur peut demander si elle est toujours vraie. Transversalement, `payments-service` est le seul à atteindre `bank_accounts`, seul `payments-service` touche `payment_tokens`, hormis une lecture restreinte au niveau des colonnes par `human_readonly`, et aucun rôle humain n'écrit par défaut.

L'append-only est structurel

Aucun rôle applicatif, aucun rôle humain et aucun grant JIT ne détient `UPDATE` ou `DELETE` sur `ledger_entries` ou `audit_events`, et cela inclut `human_breakglass`. Une correction est une nouvelle écriture, jamais une modification. Les corrections du grand livre ne sont émises que par le chemin de correction de `payments-service`, jamais par un login humain.

Il existe une exception structurelle : `ledgerly_owner`, et donc une migration en cours, peut modifier ces tables, parce que le propriétaire peut modifier ou supprimer la table et s'accorder n'importe quel privilège sur elle. C'est pourquoi le Chapitre 11 revoit chaque migration qui les touche. Le contrôle se trouve dans la revue, non dans le privilège.

Valeurs par défaut, propriété et superutilisateur

Privilèges par défaut

Beaucoup de moteurs permettent qu'un grant s'applique aux tables créées plus tard, que ce soit sous forme de privilège par défaut rattaché à un propriétaire ou de grant valable pour tout un schéma. C'est pratique au début

et un piège plus tard, parce qu'une nouvelle table hérite silencieusement de ce que dit le grant.

Ma préférence va à une valeur par défaut vide. Une nouvelle table n'est lisible par personne d'autre que le propriétaire, et la migration qui la crée énonce chaque grant dans le même fichier. Le coût est de quelques lignes de plus par migration ; le bénéfice est qu'une seule revue montre tout l'effet d'un changement sur les accès.

> **Avertissement :** Un privilège par défaut qui accorde `SELECT` sur les futures tables à un rôle large comme `human_readonly` expose la prochaine table sensible dès sa création. Revoyez ces valeurs par défaut en premier, et supprimez celles qui couvrent les futures tables.

La notation est illustrative, parce que les privilèges par défaut n'ont pas de syntaxe neutre.

```
// illustrative notation: defaults for objects created by
ledgerly_owner
DEFAULT PRIVILEGES FOR ledgerly_owner ON TABLES: none

// each migration that creates a table states its own grants
CREATE TABLE <new_table> (...);
GRANT SELECT ON <new_table> TO app_billing_api;
```

Le danger de la propriété

Dans les moteurs courants, le propriétaire peut supprimer un objet, en modifier la définition, accorder des privilèges à n'importe qui et, dans bien des cas, ignorer les restrictions au niveau des lignes qui s'appliquent aux autres. Quiconque détient la propriété peut modifier le modèle d'accès lui-même.

Les règles autour de `ledgerly_owner` sont donc strictes, et cette rigueur ne coûte pas cher. Aucun login n'existe pour lui. Les rôles applicatifs ne sont jamais propriétaires de tables, même de celles qu'ils créent pour leur propre usage, parce que « le service crée sa table au démarrage » est la manière dont un workload finit par posséder une partie du schéma. Le propriétaire est le rôle le moins utilisé du système, ce qui est l'état correct.

Le superutilisateur

La plupart des moteurs ont un rôle d'administration intégré qui contourne les vérifications de privilèges, bien que les services managés puissent le retenir. Je l'appelle ici le superutilisateur ; il se trouve en dehors du schéma de rôles de Ledgerly parce qu'il vient avec le moteur.

Je le traiterais comme un instrument sous scellés. Aucune application, aucun job et aucune session humaine ne l'utilise dans le travail quotidien. Son credential est conservé là où l'équipe plateforme peut le récupérer et où la récupération laisse un enregistrement, et il sert aux opérations au niveau du moteur qu'aucun rôle ne peut faire, comme réparer la réplication. La limite honnête est qu'il peut défaire chaque contrôle de ce chapitre, et je ne connais pas de conception qui supprime ce pouvoir sans rendre la reprise impossible. Rendez son usage rare, visible et revu ; le Chapitre 15 y revient. L'administration courante, comme appliquer les définitions de rôles ou délivrer des logins, utilise `ledgerly_access_admin` à la place. Si le superutilisateur sert à des tâches courantes, il manque à un rôle un privilège qu'il devrait avoir.

`ledgerly_access_admin` est un rôle utilisé par trois principals, chacun par son propre login : l'étape du pipeline qui applique la définition des accès, le credential broker (Chapitre 4) et l'access gateway (Chapitre 7). Il crée des logins et gère les appartenances, n'est propriétaire d'aucune table et ne détient aucun privilège sur les données, et c'est pourtant un pouvoir étendu : quiconque le détient peut accorder n'importe quoi dans les rôles qu'il administre. Des logins séparés distinguent les trois dans le log, mais pas dans ce qu'ils peuvent faire.

Ce qui borne le pouvoir propre du plan de contrôle n'est donc pas le rôle mais son environnement. Chacun des trois logins n'est joignable que depuis son propre composant. Chaque action va dans le log d'audit (Chapitre 14), et la vérification de dérive ci-dessous compare le résultat avec le code. Les modifications de la définition des accès passent par leur propre petit groupe de relecteurs. Ces limites rendent l'usage abusif visible ; elles ne le rendent pas impossible, et je le dirais lors de la revue des accès.

Restreindre davantage: vues, règles de ligne et tenants

Vues

Une **vue** est une requête stockée qui se comporte comme une table. Un rôle à qui l'on a accordé l'accès à la vue lit ce que la requête expose sans détenir de privilège sur la table sous-jacente.

```
// illustrative: a narrowed projection owned by ledgerly_owner
CREATE VIEW customer_support_summary AS
    SELECT id, created_at, status FROM customers;
GRANT SELECT ON customer_support_summary TO app_admin_console;
```

> **Remarque :** Dans beaucoup de moteurs, une vue s'exécute avec les privilèges de son propriétaire, non de l'appelant. C'est ce qui fait fonctionner la restriction, et c'est aussi pourquoi une vue qui sélectionne trop de données les laisse fuiter vers chaque rôle qui a un grant sur la vue. Certains moteurs vous laissent aussi choisir l'autre comportement. Vérifiez ce que fait le vôtre par défaut, et testez la vue avec un rôle qui ne détient aucun privilège sur la table de base.

Je préfère les grants au niveau des colonnes là où le moteur les prend en charge, parce que le grant vit à côté de la table. Une vue vaut mieux quand la forme est l'enjeu : une jointure, une valeur calculée, ou une interface stable au-dessus d'une table qui change. Le livre n'utilise pas de vue par demande pour l'accès JIT, pour deux raisons : une vue doit avoir un propriétaire, donc la gateway qui la crée aurait besoin du pouvoir du propriétaire, et une vue créée pour chaque demande casse le modèle de propriété.

Règles de ligne et tenants: une règle de décision

Une règle de ligne restreint les lignes à l'intérieur d'une cellule de la matrice et ne change pas ce que dit la cellule. Le support examine un client à la fois, si bien qu'une règle telle que « `app_admin_console` ne voit que les lignes de `invoices` où `customer_id` est égal au client du dossier » convient là où un filtre de colonnes ne convient pas. Les moteurs divergent sur la syntaxe, si bien que le livre ne montre aucune notation.

Les clients de Ledgerly sont des entreprises, et chacune s'attend à ne pas voir les factures d'une autre. Cette isolation relève surtout de l'application : `billing-api` filtre par client à chaque requête, et aucun rôle ne sépare un tenant d'un autre, parce que le rôle applicatif doit tous les lire pour en servir un seul.

Utilisez une règle de ligne, pour le support ou comme seconde couche entre tenants, seulement quand deux conditions sont réunies : une mauvaise ligne est grave, et l'application définit la valeur de contexte de façon fiable pour chaque session. Le coût est que chaque instruction porte le prédicat et que quelque chose doit définir le contexte à chaque session. Derrière un pooler, une valeur définie sur une session peut atteindre le prochain emprunteur de la même connexion (le Chapitre 5 y revient). Quand le contexte manque, la règle doit ne renvoyer aucune ligne, et c'est ce cas qu'il faut tester en premier. Une règle qui échoue en mode ouvert est pire que pas de règle du tout. Des bases de données séparées constituent une frontière plus forte, pour un prix opérationnel bien plus élevé ; laquelle convient dépend du nombre de tenants que vous avez et de ce que vous leur avez promis.

Le plafond de l'accès just-in-time

`human_jit_write` définit le plafond : le maximum qu'une demande just-in-time (JIT) peut solliciter, exprimé sous forme de privilèges sur des tables et des colonnes nommées. L'accès JIT est un accès accordé pour une tâche précise et pour une durée limitée ; le Chapitre 8 traite des approbations et des durées de vie, et ce chapitre ne définit que le plafond.

Le rôle n'est jamais accordé à un login. Une demande est une liste de triplets (privilège, table, colonnes) qui tient à l'intérieur du plafond, stockée avec la demande. À l'approbation, exactement ces triplets sont accordés directement au login de session du demandeur, et à l'expiration, exactement ces triplets sont révoqués. Un rôle partagé ne peut pas être restreint demande par demande ; le périmètre propre à chaque demande vit donc dans les grants, et le rôle trace la limite extérieure.

```
// the ceiling: what a JIT request may ask for, never granted to a
login
GRANT SELECT ON customers, payments, invoices, invoice_lines
    TO human_jit_write;
// id and created_at are assigned by the database, never changed or
supplied
// by a data fix, so the write privileges below name every other
column
GRANT UPDATE (<every column except id, created_at>)
    ON customers, payments, invoices, invoice_lines TO
human_jit_write;
GRANT INSERT (<every column except id, created_at>)
    ON customers, payments, invoices, invoice_lines TO
human_jit_write;
// DELETE only where a request names it, and never on a table with
// Restricted columns
GRANT DELETE ON customers, payments, invoices, invoice_lines
    TO human_jit_write;

// illustrative column lists; Restricted columns are not named
anywhere
GRANT SELECT (customer_id), UPDATE (customer_id)
    ON bank_accounts TO human_jit_write;
GRANT SELECT (customer_id, provider, last_four, expires_on),
      UPDATE (customer_id, provider, last_four, expires_on)
    ON payment_tokens TO human_jit_write;
// no table-level SELECT on either table: it would include the
Restricted columns
// no INSERT or DELETE on either: a new or removed row sets or
destroys
// Restricted values
```

Lisez le bloc pour ce qu'il omet. `bank_accounts` et `payment_tokens` ne sont ouverts que pour les quelques colonnes non Restricted nommées, et jamais pour les identifiants de ligne ou les horodatages : `iban`, `account_holder_name` et `token_value` ne peuvent jamais figurer dans une demande JIT. `INSERT` est limité aux quatre premières tables, et il ne peut pas nommer `id` ni `created_at`, si bien qu'une ligne restaurée ou nouvelle reçoit toujours des valeurs attribuées par la base de données. Les deux tables append-only et les enregistrements des demandes d'accès, qui appartiennent à la gateway, sont entièrement hors du plafond. Chaque triplet, sauf un `DELETE`, nomme ses colonnes.

Deux coûts sont énoncés une fois, ici. Les colonnes qui relient une ligne à son propriétaire, comme `customer_id`, ne sont pas Restricted, si bien qu'un `UPDATE` JIT peut les modifier. Un tel changement de lien de propriétaire reste à l'intérieur du plafond, mais il appartient à une classe d'opéra-

tions dangereuses : la politique d'exécution le soumet à une porte, et il est acheminé vers le chemin produit (Chapitre 10) au lieu d'être exécuté en SQL brut. La porte et le chemin produit sont le contrôle, et non la seule approbation selon le principe des quatre yeux. Et un `SELECT` JIT affiche sans masquage toute colonne Confidential qu'il nomme, ce qui fait partie de ce que l'approbateur accepte.

Ce qui garde le plafond honnête, c'est une vérification, non la confiance. La gateway refuse toute demande qui le dépasse (le Chapitre 8 montre la vérification), et le test de dérive ci-dessous affirme qu'aucun login n'est membre de `human_jit_write`, la défaillance qui transformerait un plafond en grant permanent.

Quand le moteur n'a pas de grants au niveau des colonnes

Tous les moteurs ne prennent pas en charge les grants au niveau des colonnes. Le livre utilise exactement une solution de repli : là où le moteur ne peut pas accorder de privilèges sur des colonnes, les colonnes Restricted sont déplacées dans des tables séparées, `bank_account_secrets` et `payment_token_values`.

```
table bank_account_secrets:      // owned by ledgerly_owner
    bank_account_id     text primary key
    iban                text not null
    account_holder_name text not null

table payment_token_values:      // owned by ledgerly_owner
    payment_token_id    text primary key
    token_value         text not null

// reachable by app_payments_service and human_breakglass only;
// outside the ceiling, never part of a JIT request
```

Avec la solution de repli, les tables principales ne contiennent aucune colonne Restricted, si bien qu'un grant JIT au niveau de la table sur elles ne peut pas atteindre de données Restricted. Le coût : le détail de la demande descend au niveau de la table, et l'approbateur a approuvé chaque colonne non Restricted de la table, y compris les colonnes Confidential, et pas seulement celles que le demandeur a nommées.

Break-glass et les opérations au-dessus du schéma

`human_breakglass` est le seul rôle humain qui lit les colonnes Restricted. Il lit toutes les tables et écrit en `INSERT` et `UPDATE` sur toutes les tables sauf les deux tables append-only ; sur `bank_accounts` et `payment_tokens`, il n'écrit qu'en `UPDATE`, et uniquement sur les colonnes non Restricted. Il sert aux urgences, non aux corrections courantes, et il est absent de la matrice parce que seule une session le détient, pour une courte fenêtre ; aucun workload permanent ne le détient.

Les colonnes Restricted ne sont modifiées que par un chemin applicatif. Un humain ne les écrit jamais en SQL brut, que ce soit dans une session JIT ou autrement : `admin-console` le demande à `payments-service`, le seul à y écrire, après l'approbation d'une seconde personne, et la modification est enregistrée dans `audit_events`. Le Chapitre 10 décrit ce chemin. Ces faits marquent la limite de ce que les grants feront jamais. Un grant dit quelles tables et quelles colonnes une session peut toucher ; il ne peut pas dire quelle instruction, combien de lignes, ni si ce changement précis est celui qui a été approuvé. C'est le rôle de la partie Exécution, et le Chapitre 9 explique comment la gateway vérifie chaque instruction. Savoir où s'arrêtent les grants vous aide à décider où placer une nouvelle capacité.

Les rôles comme code, et la dérive

Un schéma de rôles qui n'existe que dans la base de données est un schéma que personne ne peut revoir. Les grants ci-dessus sont assez courts pour être lus, et ils doivent vivre dans le contrôle de version, changer par la revue et être appliqués par le pipeline, comme le schéma qu'ils protègent.

Où vivent les définitions

Je garderais deux types de fichiers. Les migrations créent les tables et énoncent les grants de chaque table. Une définition des accès distincte et revue liste les rôles, les appartenances entre rôles et le plafond. Le pipeline applique les deux en étapes séparées. La définition des accès est appliquée par le propre login de l'étape du pipeline dans `ledgerly_access_admin`, le même rôle que le broker et la gateway utilisent par leurs propres logins ; il ne détient aucun privilège sur les données. Créer des rôles est un

pouvoir plus fort que modifier des tables ; son étape doit donc avoir son propre petit groupe de relecteurs. La question du relecteur n'est pas de savoir si le SQL est valide, mais si ce grant est le plus étroit qui fait fonctionner le changement. Un relecteur qui a la matrice sous les yeux peut y répondre en une minute.

Restreindre un système existant

Ledgerly part d'un seul login partagé qui possède tout. Je ne passerais pas de là à la matrice en un seul changement, parce qu'un grant trop étroit casse la production et que l'équipe ne peut pas savoir ce qu'elle a oublié tant que ça n'échoue pas. Un chemin en quatre étapes :

1. **Donnez d'abord à chaque workload son propre login, avec les mêmes privilèges qu'avant.** Rien ne casse, et le log commence immédiatement à nommer le workload.
2. **Observez.** Utilisez les instructions enregistrées sous chaque login pour lister ce que le workload touche réellement, sur une période assez longue pour inclure le job mensuel lent. Un log des instructions peut contenir des valeurs littérales ; traitez-le donc comme sensible : il peut contenir des données client ou des valeurs Restricted collées dans une requête. Restreignez qui le lit, fixez sa durée de conservation, et préférez journaliser la forme des instructions avec des paramètres là où le moteur le propose.
3. **Définissez le rôle étroit et déplacez-y le login**, d'abord dans un environnement de staging qui exécute les mêmes jobs, puis en production en gardant les anciens privilèges sous la main pendant une courte période.
4. **Retirez les anciens privilèges** et surveillez les instructions refusées.

Le coût se compte en temps calendaire, non en effort. Les équipes ont tendance à sauter la deuxième étape, et le résultat prévisible est un chemin rarement utilisé qui casse des mois plus tard.

La dérive

La **dérive** est l'écart entre l'accès que déclare le code et l'accès que détient la base de données. Elle apparaît quand un problème est corrigé la nuit avec un `GRANT` écrit à la main : le grant fonctionne, l'incident se termine et

personne ne le note. La vérification compare le déclaré et le réel selon une planification, et elle doit échouer bruyamment quand elle ne peut pas lire l'un des deux côtés.

```
function checkRoleDrift():
    declared = loadDeclaredAccess()
    try:
        actual = readActualAccess()
        requests = loadAccessRequests()
        known = loadGatewayLogins()
    on ReadFailed:
        // an unreadable state is not a clean state
        notify("platform", "role drift check could not read a source")
        raise DriftCheckFailed

    for each extra in difference(actual.roleGrants,
declared.roleGrants):
        notify("platform", "grant not in code: " + extra)
    for each login in actual.loginsOwningObjects():
        notify("platform", "login owns objects: " + login)
    for each login in actual.membersOf("human_jit_write"):
        notify("platform", "login is a member of the ceiling: " +
login)
    for each login in actual.allLogins():
        if login is human and login not in known:
            notify("platform", "login unknown to the gateway: " +
login)
        // memberships must match what the login was issued for
        for each role in difference(actual.rolesOf(login),
expectedRoles(login)):
            notify("platform", "unexpected membership: " + role + "
for " + login)
        // a direct grant is legitimate only as part of a live request
        live = unexpiredTriples(requests, login)    // empty for
service logins
        for each g in difference(actual.directGrantsOf(login), live):
            notify("platform", "direct grant without a live request: "
+ g + " to " + login)
    for each login in actual.membersOf("human_breakglass"):
        if not hasOpenBreakGlass(requests, login):
            notify("platform", "break-glass member without a request:
" + login)
```

`expectedRoles(login)` vaut `human_readonly` pour un login de session de la gateway, plus `human_breakglass` tant qu'une demande break-glass est ouverte, et l'unique rôle `app_*` pour lequel le broker a délivré le login. La vérification détecte donc un `GRANT app_payments_service TO <session login>` ou un `GRANT ledgerly_owner TO <login>` écrit à la main, aussi bien qu'un grant direct de privilège.

Ici, `roleGrants` ne couvre que les appartenances de rôle à rôle et les grants de privilège à un rôle, que le code déclare. Les logins ne sont pas dans le code : ils vont et viennent avec les sessions. Les logins humains sont vérifiés par rapport aux enregistrements de la gateway. Les grants directs de chaque login, logins de service compris, sont comparés aux triplets `access_request_privilege` qui n'ont pas expiré. Un login de service n'a pas de demandes, si bien que tout grant direct sur un tel login est signalé ; c'est ainsi qu'un `GRANT` d'un privilège écrit à la main la nuit pour un login de service est remarqué. Un grant qui a survécu à sa demande, ou un membre break-glass sans demande ouverte, est signalé même quand le job de révocation annonce un succès. L'accès JIT fonctionne par grants directs, et l'expiration par un job de révocation ; si le job s'arrête, les grants restent, et cette vérification le remarque indépendamment du job (le Chapitre 8 traite de ce couplage). La vérification ne lit que les grants et les appartenances. Elle ne voit pas ce qu'une session en fait, ce qui relève de nouveau de la partie Exécution.

La vérification ne répare rien, parce qu'une révocation automatique de grants inconnus en production est une nouvelle façon de casser la production la nuit. Elle ne dit pas non plus si l'état déclaré est le bon : un grant erroné, correctement déclaré, passe. Ce jugement appartient à la revue des accès du Chapitre 14.

Compromis et coûts

Chaque contrôle de ce chapitre a un prix, et ce prix est payé par des personnes précises.

Les rôles par workload coûtent de la maintenance. Cinq rôles signifient cinq définitions à garder en phase avec cinq bases de code. Quand un service a besoin d'une nouvelle table, quelqu'un met à jour le rôle dans le même changement ; si le grant arrive après le code, le déploiement échoue. L'atténuation consiste à faire du grant une partie de la migration. Avec un service et deux ingénieurs, ce surcoût n'est pas nul, et un schéma plus petit, composé d'un propriétaire, d'un rôle applicatif et d'un rôle en lecture seule, est une première étape raisonnable.

Les grants étroits provoqueront des instructions refusées. Certains refus sont des erreurs dans les grants, et d'autres du code qui est allé plus loin qu'il n'aurait dû. Quiconque est d'astreinte paie pour les deux quand un refus apparaît pendant un incident. Faire tourner le staging avec les mêmes rôles que la production réduit la surprise ; ignorer le coût pousse les équipes à élargir un grant dans la précipitation, et c'est par la précipitation que commence la dérive.

Les grants au niveau des colonnes alourdissent la revue. Chaque liste de colonnes doit être maintenue quand la table change. Une nouvelle colonne sensible est exposée par défaut sous un grant au niveau de la table ; pour les tables sensibles, j'accorde donc sur des colonnes nommées et j'accepte les lignes supplémentaires. Là où le moteur n'a pas cette fonctionnalité, la solution de repli coûte des jointures, des tables supplémentaires à migrer et une approbation JIT plus grossière.

Les vues et les règles de ligne ajoutent des pièces mobiles. Elles valent leur poids quand la forme ou l'isolation est l'enjeu, et sont un mauvais compromis quand on les ajoute pour se sentir prudent. Une règle de ligne dont le contexte peut manquer relève du théâtre de sécurité : elle ajoute de la friction à chaque requête et échoue en mode ouvert dans le cas pour lequel elle a été écrite.

La discipline de propriété ralentit certaines personnes. Les ingénieurs habitués à créer une table dans une console doivent désormais passer par une migration. La friction est voulue, et elle vaut moins sur une base de développement jetable que sur la production. Appliquez d'abord le modèle à la production, et ne l'assouplissez ailleurs que lorsque vous pouvez dire ce à quoi vous renoncez.

Quand ne pas utiliser autant de structure. Un prototype avec un seul développeur et aucune donnée client n'a pas besoin du schéma de rôles complet. La séparation entre propriétaire, rôle applicatif et lecture seule suffit jusqu'au deuxième service ou à la première colonne sensible. Faites grandir le schéma quand « qui a exécuté ça ? » ne trouve pas de réponse dans le log, ou quand le credential d'un workload pourrait faire plus de dégâts que sa tâche ne le justifie.

Erreurs courantes

- **Traiter le rôle applicatif comme le propriétaire.** C'est tentant, parce que les migrations s'exécutent alors au démarrage avec une étape de moins. Le service détient alors le pouvoir de supprimer et de réaccorder. Utilisez un migrator distinct et un propriétaire sans login.
- **Accorder en fonction des requêtes de l'an dernier.** Un rôle copié sur le login existant le plus large garde chaque privilège qui a un jour été pratique. Dérivez le rôle de ce que fait le workload maintenant, observé sous son propre login.
- **Ajouter un rôle par personne.** Les rôles sont des ensembles de privilèges, et les logins sont par personne. Un rôle nommé d'après un individu transforme le schéma en liste du personnel. Gardez des rôles centrés sur les tâches.
- **Utiliser un seul rôle pour plusieurs services parce qu'ils sont « similaires ».** Les services divergent, et le rôle garde l'union de ce dont chacun a besoin. Un rôle par service coûte quelques lignes.
- **Laisser des valeurs par défaut pour les futures tables.** Un privilège par défaut économise une ligne dans chaque migration et expose la prochaine table sensible à un rôle large. Gardez la valeur par défaut vide et accordez explicitement.
- **Accorder le rôle plafond à un login le temps d'une semaine chargée.** Le plafond devient alors un grant permanent avec un nom sympathique. Gardez le plafond détaché : seule la vérification des demandes le lit, et le test de dérive signale toute appartenance.
- **Cacher une colonne sensible derrière une vue en laissant la table lisible.** La vue ne restreint rien pour un rôle qui peut atteindre la table de base. Révoquez sur la table et testez avec un rôle qui ne détient aucun privilège sur la table.
- **Faire confiance à un schéma déclaré que personne ne compare.** Des rôles dans le contrôle de version n'empêchent pas le `GRANT` de 2 heures du matin. Comparez le déclaré et le réel selon une planification, et faites échouer bruyamment la comparaison quand elle ne peut pas lire.
- **Utiliser le superutilisateur pour le travail courant.** Il fonctionne à chaque fois, et c'est pourquoi son usage se répand. Quand une tâche courante en a besoin, ajoutez le privilège manquant au bon rôle.

Points clés

- Un login partagé détruit à la fois la preuve et le périmètre : le log ne peut pas dire qui a agi, et le login doit être autorisé à faire tout ce que fait son appelant le plus large.
- Gardez l'identité, le login et le rôle séparés : un principal a une identité, se connecte par un login, et le login est membre de rôles ; personne ne se connecte en tant que rôle.
- `ledgerly_owner` est propriétaire de tout et n'a pas de login, `ledgerly_migrator` ne l'endosse que pendant une migration, et chaque workload reçoit son propre rôle `app_*` dimensionné pour sa tâche.
- Faites partir chaque rôle d'aucun privilège, utilisez des grants au niveau des colonnes là où des colonnes sensibles côtoient des colonnes ordinaires, gardez les privilèges par défaut vides, et ne vous rabattez sur des tables de secrets séparées que là où le moteur ne peut pas accorder de privilèges sur des colonnes.
- `human_jit_write` est un plafond qui n'est jamais accordé à un login : chaque demande approuvée en accorde un sous-ensemble directement au login de session, et aucune demande ne peut inclure de colonne Restricted.
- Les rôles et les grants ont leur place dans le contrôle de version et dans des étapes de pipeline revues, et une vérification de dérive planifiée, qui échoue bruyamment quand elle ne peut pas lire la base de données, maintient ensemble l'état déclaré et l'état réel.
- Chaque niveau coûte quelque chose à quelqu'un : de la maintenance pour les rôles, des instructions refusées pour les grants étroits, une charge de revue pour les listes de colonnes, et une approbation plus grossière là où le moteur n'a pas de grants au niveau des colonnes.

Les rôles disent ce qu'un principal peut faire une fois connecté, mais ils ne disent encore rien de la manière dont un service prouve son identité. Le Chapitre 4 franchit l'étape suivante : comment les credentials parviennent aux workloads, pourquoi les credentials dynamiques et de courte durée changent la forme du problème, et ce qui remplace le mot de passe dans le fichier de configuration.

PARTIE II

L'accès des applications

Chapitre 4 — Credentials et secrets

Imaginez un vendredi après-midi chez Ledgerly. Une personne de l'équipe backend est partie dans un autre groupe, et le contact sécurité demande, poliment, si le mot de passe admin partagé peut être changé. Alex, de l'équipe plateforme, convient qu'il devrait l'être. Puis Alex commence à dresser la liste des endroits où vit le mot de passe : la configuration des cinq services, les variables du pipeline, l'étape d'export de `reporting-job`, un gestionnaire de mots de passe d'équipe et les connexions enregistrées dans plusieurs clients de bureau.

Le changer dans la base de données prend une seconde. Le changer partout, au même moment, implique un redémarrage coordonné de tous les services pendant les heures ouvrées, et Robin, d'astreinte ce week-end, pose la question légitime : qu'advient-il des factures en cours de traitement ? Le changement est reporté à une « fenêtre calme » qui n'arrive jamais. Le mot de passe survit à la personne, puis à la suivante.

Ce chapitre répond à la question soulevée ce vendredi-là : comment un service obtient-il un credential, et comment en changer un sans panne ? Il ajoute l'étape suivante de l'état cible de l'exemple fil rouge. Les credentials des services deviennent de courte durée ou dynamiques, et ils parviennent aux workloads par leur identité plutôt que par un fichier partagé.

Les credentials relèvent de la partie Identité du modèle en cinq parties, et ils sont nécessaires sans être la réponse. La thèse du livre est que « L'accès aux bases de données en production n'est pas un problème de distribution de credentials ; c'est un problème d'autorisation, d'exécution et d'imputabilité. » Une meilleure gestion des credentials réduit le nombre de ceux qui peuvent obtenir un login. Elle ne dit rien de ce qu'un login peut faire une fois entré, ni de ce qu'il a fait, et ce sont là les tâches des chapitres sur les grants, l'exécution et les enregistrements.

Le problème, et pourquoi il est difficile

Un **credential** est ce qu'un principal présente pour s'authentifier ; un **secret** est toute valeur qui donne un accès si elle est divulguée. Un mot de passe est les deux à la fois. Le Chapitre 3 a donné à chaque service son propre rôle, de sorte qu'un login dise ce qu'il peut faire. Ce chapitre traite de l'autre moitié : comment le login prouve qui il est, et combien de temps cette preuve reste valable.

Un mot de passe a un cycle de vie même quand personne n'en conçoit un. Il est créé, copié partout où un service peut le lire, utilisé pendant des mois, et finalement changé ou oublié. Chaque copie est un endroit où une personne ou un processus peut le lire, et la question « qui peut voir ce mot de passe ? » a autant de réponses qu'il y a de copies. Dans l'état initial, la réponse est « à peu près toute l'équipe backend, et tout ce qui peut lire la configuration d'un service ».

Le remède évident est un meilleur mot de passe : plus long, aléatoire, stocké dans un endroit mieux rangé. Cela aide contre la devinette, et la devinette n'a jamais été le risque principal. Le risque, c'est le nombre d'endroits où se trouve la valeur, et le fait qu'aucun d'eux ne peut être vidé sans toucher aux autres. Un mot de passe robuste dans cinq fichiers de configuration reste cinq copies.

La seconde difficulté est la **rotation**, c'est-à-dire le remplacement d'un credential par un nouveau, selon un calendrier ou sur soupçon. La rotation est difficile parce qu'un credential est un état partagé entre deux systèmes : la base de données connaît la valeur, et chaque consommateur aussi. Changer un côté d'abord casse l'autre. Une équipe qui ne peut pas changer un mot de passe sans panne cessera, raisonnablement, de le changer, et une politique de rotation que personne ne peut exécuter ajoute une entrée au calendrier sans supprimer aucune exposition.

Ce que les credentials ne peuvent pas faire

Prenez un login du job de reporting, valable une heure, qui a fuité dans un canal de discussion. C'est un bon credential selon tous les critères de ce chapitre : il est unique, de courte durée et obtenu par identité. Pendant son heure, il peut toujours lire les trois colonnes de `customers` accordées au rôle (`id`, `created_at` et `status`), sur chaque ligne, car c'est ce que

`app_reporting_job` a le droit de lire. Les durées de vie courtes réduisent la fenêtre ; elles ne réduisent pas le rôle. Trois choses bornent une fuite, et aucune n'est le credential lui-même : le **rôle** qu'il porte (Chapitre 3), la **portée réseau** qui décide d'où il peut être utilisé (Chapitre 7), et le **log** qui enregistre ce qu'il a fait (Chapitre 14). Un login qui ne peut être présenté que depuis le réseau du pooler, avec un rôle qui ne peut pas voir les colonnes Restricted, fait un incident plus petit que ce que n'importe quelle politique de mots de passe peut obtenir.

Il en va de même pour la conception la plus robuste ci-dessous. Un broker, un lease et la workload identity répondent tous à la question de savoir qui peut obtenir un login. Ils ne décident pas de ce que le login peut faire, et ils n'empêchent pas un login autorisé d'exécuter une instruction nuisible.

C'est une question d'ingénierie, pas de politique. Une règle qui dit « faire une rotation tous les 90 jours » n'a rien à dire sur la manière dont les cinq services apprennent la nouvelle valeur au même instant. La réponse se trouve dans la manière dont les credentials sont émis, livrés et remplacés, ce qui est une conception que l'équipe doit construire.

La vie d'un credential

Il est utile de suivre un credential de sa naissance à sa retraite et de se demander, à chaque étape, qui peut le lire et ce qui le révélerait. Les endroits habituels sont les suivants.

Endroit	Qui peut le lire	Comment il fuit en général	Meilleur emplacement
Fichier de configuration dans le dépôt	Toute personne ayant accès au dépôt, pour toujours, via l'historique	Copié dans un fork, une discussion, un ticket	Un gestionnaire de secrets, référencé par son nom
Variable d'environnement	Le processus, ses enfants, et quiconque peut inspecter le processus ou ses crash dumps	Affiché par un endpoint de debug ou un rapport d'erreur	Un fichier ou de la mémoire remis au démarrage, avec des permissions de fichier étroites
Image de conteneur ou artefact de build	Toute personne qui peut récupérer l'image	Intégré au moment du build, donc chaque copie le porte	Rien dans l'image ; récupéré au démarrage

Endroit	Qui peut le lire	Comment il fuit en général	Meilleur emplacement
Variable CI/CD et log de job	Les administrateurs du pipeline ; parfois chaque job	Affiché par un script, imprimé par une commande en échec	Valeur de courte durée récupérée par l'identité propre du job
Gestionnaire de mots de passe d'équipe	Toute personne ayant accès au coffre partagé	Partagé par habitude ; aucune trace de qui l'a lu	Accès individuel avec trace, pour les rares qui restent
Connexion enregistrée d'un client de bureau	Quiconque utilise cet ordinateur portable	Ordinateur portable perdu ou sauvegardé	Aucun credential enregistré ; credentials de session fournis par une gateway (Chapitre 7)

Deux choses ressortent du tableau. D'abord, les fuites ne sont presque jamais astucieuses. Ce sont des copies faites par commodité : une valeur collée dans un ticket pour qu'un collègue puisse reproduire un problème, une variable affichée pendant le débogage d'un déploiement. Ensuite, la colonne « meilleur emplacement » donne presque toujours la même réponse : la valeur vit dans un seul endroit géré, et les consommateurs la récupèrent quand ils en ont besoin.

> **Avertissement :** Un secret qui est apparu dans une couche d'image, un historique de dépôt ou un log de job doit être considéré comme divulgué, même si personne ne l'a lu. Supprimer la ligne ne supprime pas les copies. Le remède est de remplacer le credential, et c'est pourquoi une équipe a besoin d'un chemin de rotation qui fonctionne avant le jour où il devient nécessaire.

Un secret a encore une propriété qui mérite d'être nommée : la durée pendant laquelle il reste valable. Supposons qu'un mot de passe soit valable trois ans : il porte trois ans d'exposition pour chaque copie jamais faite. Une vieille copie d'un credential qui dure une heure ne vaut rien. Les durées sont données à titre d'illustration. C'est pourquoi la durée de vie compte davantage que la robustesse.

Les gestionnaires de secrets et la manière dont un service lit un secret

Un **gestionnaire de secrets** est un service qui stocke des secrets, contrôle qui peut lire chacun d'eux et enregistre chaque lecture. HashiCorp Vault et les stores de secrets des grands fournisseurs cloud en sont des exemples. Le pattern compte plus que le produit : un seul endroit détient la valeur, l'accès est décidé par principal, et une lecture laisse une trace.

Déplacer le mot de passe dans un gestionnaire de secrets ne réduit pas à lui seul le nombre de lecteurs. Cela change qui ils sont et donne un nom à chaque lecture : une identité de service précise lit un chemin précis, et le gestionnaire de secrets dit quand. C'est un vrai gain pour l'investigation et la révocation, et un gain modeste pour l'exposition.

Le premier secret

Un service a besoin de quelque chose pour prouver qui il est au gestionnaire de secrets avant de pouvoir lire quoi que ce soit. Si ce quelque chose est un autre mot de passe stocké, le problème est monté d'un niveau et rien n'est résolu. C'est la question de l'amorçage, parfois appelée le problème du premier secret.

La réponse est la **workload identity** : une identité que la plateforme donne à un workload en cours d'exécution, comme une instance de service ou un job, pour qu'il puisse s'authentifier sans mot de passe stocké. La plateforme qui démarre le workload se porte garante de lui, et le gestionnaire de secrets fait confiance à cette déclaration. Le service ne détient jamais de valeur de longue durée qui lui soit propre pour cette étape. Je considère la workload identity comme ce qui rend le reste de la conception possible, parce qu'elle met fin à la régression.

Lire au démarrage, et lire à nouveau

Un service a deux moments raisonnables pour lire son credential de base de données : une fois au démarrage, ou chaque fois qu'il ouvre une connexion. Lire une seule fois est simple, et cela fonctionne avec des credentials statiques qui changent rarement. Cela échoue quand le credential change, car le processus en cours conserve l'ancienne valeur jusqu'au redé-

marrage. Lire au moment de la connexion, au moyen d'un client qui interroge à nouveau le gestionnaire de secrets après un échec d'authentification, permet de supporter la rotation sans redémarrage.

```
function connectWithCredential(service):
    attempts = 0
    while attempts < maxAttempts:
        try:
            slot = readSecret(path: "db/" + service + "/active")
            credential = readSecret(path: "db/" + service + "/" +
slot)
            return openConnection(host: "db-pooler.internal.example",
                                  login: credential.login,
                                  password: credential.password)
        on AuthenticationFailed:
            // the value may have rotated since the last read; read
again
            attempts = attempts + 1
        on SecretStoreUnavailable:
            // fail the start; never fall back to a value baked into
the image
            raise StartRefused
    raise StartRefused
```

Le bloc montre les deux défaillances qui comptent. Un échec d'authentification déclenche une nouvelle lecture, de sorte qu'une rotation ne devient pas une panne. Un gestionnaire de secrets injoignable fait refuser le démarrage, car se rabattre sur une copie de secours recréerait le fichier que cette conception a supprimé. Ce refus a un prix, et la section Compromis et coûts y revient.

> **Remarque :** Le gestionnaire de secrets est lui-même une porte. Qui peut lire le chemin `db/billing-api`, qui peut le modifier et si les lectures sont journalisées sont les mêmes questions que pour la base de données. Donner à chaque service un accès en lecture à chaque chemin reconstruirait le mot de passe partagé avec une étape de plus.

La rotation sans panne

Le remède à « changer un côté d'abord casse l'autre » est de ne jamais avoir de moment où une seule valeur fonctionne. C'est le **pattern à deux credentials**, souvent appelé rotation A/B. Chaque service a deux logins, A et

B, tous deux membres du même rôle `app_*`. À tout moment, l'un est actif et l'autre inactif.

Pour faire une rotation, on change le login inactif, on publie la nouvelle valeur, on demande aux consommateurs de basculer, on attend que l'ancien login ne soit plus utilisé, puis on l'invalide. À aucun moment un consommateur ne détient une valeur que la base de données a cessé d'accepter. En pseudocode, avec les points de défaillance marqués :

```
function rotateServiceCredential(service):
    active = readSecret(path: "db/" + service + "/active")   // "a" or
"b"
    idle = otherSlot(active)
    if openSessions(loginFor(service, idle)) > 0:
        // an earlier rotation did not finish; changing this login
cuts live sessions
        raise RotationIncomplete
    newPassword = generatePassword()
    // the idle login has no users, so changing it breaks nothing
    setLoginPassword(login: loginFor(service, idle), password:
newPassword)
    try:
        writeSecret(path: "db/" + service + "/" + idle,
                    value: { login: loginFor(service, idle), password:
newPassword })
        writeSecret(path: "db/" + service + "/active", value: idle)
    on SecretStoreUnavailable:
        // the active login is untouched; the idle one holds a value
nobody can read
        notify("platform", "rotation stopped before publish: " +
service)
        raise RotationAborted
    // consumers re-read on their next connection; wait for the old
login to drain
    waitUntil(openSessions(loginFor(service, active)) == 0, deadline:
drainDeadline)
    if openSessions(loginFor(service, active)) > 0:
        // the old value still works; the rotation is not finished
        notify("platform", "old login still in use after rotation: " +
service)
        raise RotationIncomplete
    setLoginPassword(login: loginFor(service, active), password:
generatePassword())
    auditLogAppend({ action: "rotate", subject: service, actor:
"rotation-job" })
```

Trois détails portent la conception. Premièrement, la base de données et le gestionnaire de secrets sont des systèmes distincts sans transaction partagée, si bien que c'est l'ordre des étapes qui limite une défaillance : la va-

leur en service n'est jamais changée avant que sa remplaçante soit lisible. Une défaillance à n'importe quelle étape laisse le credential précédent fonctionnel. Deuxièmement, la vérification de vidange. La rotation n'est terminée que lorsque plus rien n'utilise l'ancien login, et un service qui ne relit jamais son secret est découvert à ce moment-là, pas lors d'une panne trois semaines plus tard. Troisièmement, la dernière étape écrase l'ancien login avec une valeur que personne ne conserve, mais elle ne s'exécute qu'après la vidange. Tant qu'une session détient encore l'ancien login, l'ancienne valeur fonctionne.

Le coût est que chaque service a désormais deux logins et un job qui connaît les deux, et que le consommateur doit relire le secret après un échec. Un service qui lit son credential une seule fois au démarrage a besoin d'un redémarrage progressif pour la rotation, et c'est à ce redémarrage qu'une rotation A/B redevient un événement du calendrier. Faire la rotation du mot de passe admin partagé dans l'état initial est difficile pour cette raison : il n'y a pas de second login, et aucun consommateur ne peut relire quoi que ce soit.

Je ferais la rotation selon un calendrier que l'équipe peut tenir plutôt que selon le plus court qu'elle puisse écrire. Une fuite suspectée suit un autre chemin, décrit dans « Quand un credential fuit ».

Credentials dynamiques et de courte durée

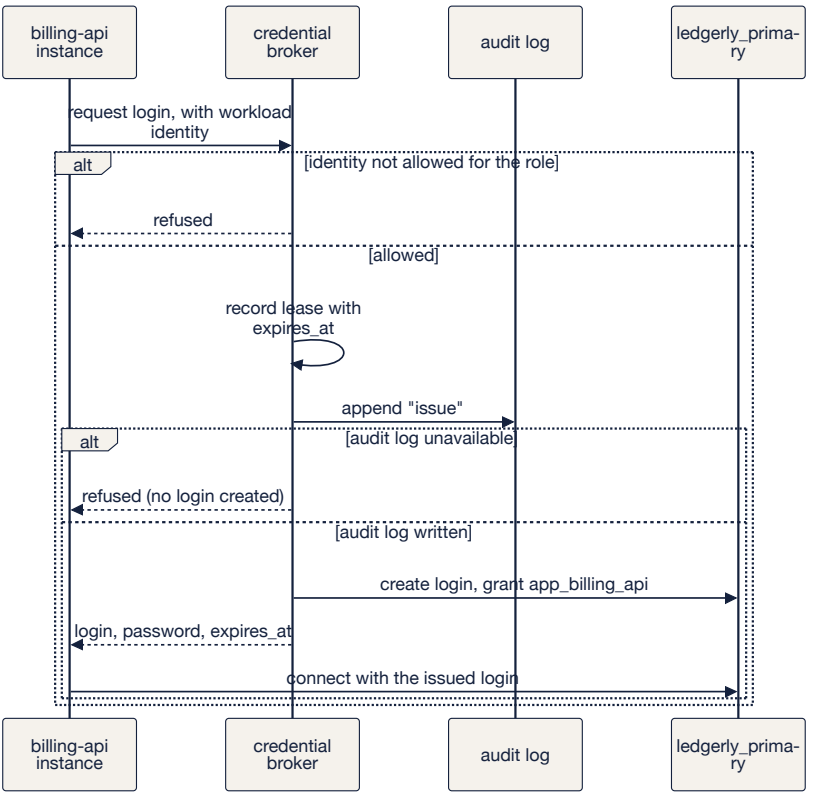
Un credential statique soumis à rotation garde une longue durée de vie entre deux rotations. L'étape suivante consiste à rendre cette durée de vie assez courte pour que la rotation cesse d'être un événement. Un **credential de courte durée** expire de lui-même dans un délai court et défini. Un **credential dynamique** est un credential de courte durée généré à la demande pour un principal, de sorte que deux demandes n'obtiennent jamais la même valeur. Tous les credentials dynamiques sont de courte durée ; tous les credentials de courte durée ne sont pas dynamiques.

Le mécanisme habituel est un **credential broker** : un service qui, à la demande, crée un login dans la base de données, lui accorde l'appartenance au bon rôle `app_*` et remet le credential du login au workload demandeur

avec un **lease**. Un lease est un enregistrement qui dit qui détient le credential, pour quel rôle et jusqu'à quand. Quand le lease expire, ou que quelqu'un le révoque, le broker supprime le login.

Chez Ledgerly, le broker se connecte avec son propre login, membre de `ledgerly_access_admin`, le rôle qui crée et supprime les logins et gère les appartenances aux rôles. Il ne possède aucune table, n'est pas le propriétaire, et personne ne l'utilise pour travailler sur les données. C'est un pouvoir plus étroit que la propriété, et il reste large : quiconque détient le rôle peut accorder n'importe quoi à l'intérieur des rôles qu'il administre, rôles d'application et rôles humains compris. Le broker n'a besoin que d'une petite partie de ce pouvoir. Lui en donner moins impliquerait de scinder le rôle, ce qui a ses propres coûts ; ici, le broker utilise le rôle tel quel. Le rôle est un seul rôle utilisé par trois principals, chacun par son propre login : l'étape du pipeline, ce broker et la gateway. Le Chapitre 3 dit ce qui borne ce pouvoir. Pour le broker, le garde-fou est la seule déclaration faite ici : son credential est détenu comme celui du propriétaire, seule sa propre workload identity peut le récupérer, et chaque action qu'il effectue va au log d'audit (Chapitre 14).

La séquence ci-dessous montre une instance de `billing-api` qui demande un login. Repérez les deux endroits où la demande est refusée : avant que quoi que ce soit n'existe, et quand le log d'audit est injoignable.



En pseudocode, c'est l'ordre des opérations qui garde la séquence correcte. Les noms de fonctions comme `createLogin` sont donnés à titre d'illustration, comme ailleurs dans le livre.

```
function issueServiceLogin(request):
    identity = verifyWorkloadIdentity(request.proof)
    if request.role not in rolesAllowedFor(identity):
        raise IssueRefused
    password = generatePassword()
    login = identity.workload + "-" + randomSuffix()
    expiresAt = now() + ttlFor(request.role)
    // the lease comes first: the sweeper reads it, so no login can
exist
    // without a record of when it must end
    leaseId = recordLease({ login: login, identity: identity.workload,
                            instance: request.instance, role:
request.role, expires_at: expiresAt })
    try:
        auditLogAppend({ action: "issue", subject: login,
                         actor: "credential-broker", lease: leaseId })
    on AuditLogUnavailable:
        markLeaseVoid(leaseId)
        raise IssueRefused
    // a failure here leaves a lease with no login; dropping a missing
login is harmless
    createLogin(login, password: password, valid_until: expiresAt)
    grant(request.role, to: login)
    return { login: login, password: password, expires_at: expiresAt }
```

Le lease est écrit avant que le login n'existe, et l'entrée d'audit est écrite avant que le login ne soit créé. Si la base de données échoue après l'enregistrement du lease, le sweeper trouve un lease sans login et le traite comme déjà révoqué. Si le log d'audit est indisponible, aucun credential n'est émis. C'est un choix délibéré, et il signifie qu'un log d'audit injoignable empêche de nouvelles instances de démarrer, un coût que le Chapitre 14 examine sous l'autre angle.

Expiration et révocation

Une expiration empêche seulement les nouvelles connexions de s'authentifier. Une session déjà ouverte continue de tourner après l'expiration du mot de passe, si bien que l'expiration comporte deux parties : le login est rendu invalide, et les sessions ouvertes sur ce login sont fermées. C'est aussi la différence entre la rotation et la **révocation**, qui fait cesser immédiatement de fonctionner un credential ou un grant. La rotation remplace ; la révocation met fin.

```
function revokeExpiredLeases():
    for each lease in expiredLeases(before: now()):
        try:
            closeSessions(login: lease.login)
            dropLogin(lease.login)
        on LoginNotFound:
            // the login is already gone, which is the goal
        on RevokeFailed:
            // an expired credential that still works is an incident
            notify("platform", "expired login still active: " +
lease.login)
            continue
        markLeaseRevoked(lease.id)
        try:
            auditLogAppend({ action: "revoke", actor: "lease-sweeper",
                             subject: lease.login, lease: lease.id })
        on AuditLogUnavailable:
            // the revoke is done; the log catches up from the retry
queue
            queueForRetry(lease.id)

function checkOverdueLeases():
    // scheduled separately, so one failing job cannot hide the other
    if count(expiredLeases(before: now() - graceMinutes)) > 0:
        notify("platform", "leases past expiry and not revoked")
    if now() - lastSuccessfulRun("revokeExpiredLeases") > maxRunAge:
        notify("platform", "lease sweeper has not completed recently")
```

Le renouvellement suit la même discipline. Une instance en cours d'exécution demande au broker de prolonger son lease avant qu'il ne prenne fin, et le broker n'accepte que tant que le lease est actif et que la durée de vie totale reste sous un plafond. Sans plafond, le renouvellement transforme un credential de courte durée en un credential de longue durée qui se trouve simplement réémis. La durée de vie elle-même relève du jugement. Je ne connais pas le bon chiffre pour votre système, et il dépend de trois choses : combien de temps une instance tourne entre deux redémarrages, combien de connexions chaque instance détient, et combien de temps prend un renouvellement quand le broker est sous charge. À titre d'illustration seulement, pensez à un lease d'une heure renouvelé toutes les trente minutes.

Le gain par rapport à la rotation A/B est qu'un credential qui a fuité expire de lui-même. Le login nomme l'instance, puisqu'il a été créé pour elle, de sorte que le log de la base de données répond à « qui a exécuté cette instruction ? » jusqu'au niveau du workload. Cela tient la promesse du Chapitre 3 : un login qui correspond à une seule instance de workload.

Workload identity et authentification à la base de données fondée sur l'IAM

Il existe une façon de supprimer complètement le mot de passe stocké. Certaines plateformes permettent à la base de données de faire confiance au propre système d'identité de la plateforme : le workload demande à la plateforme un token de courte durée, et la base de données accepte ce token à la place d'un mot de passe. Les fournisseurs cloud proposent cela sous des noms comme IAM database authentication, et certaines bases de données peuvent vérifier des tokens émis par un fournisseur d'identité externe.

L'intérêt est qu'il n'y a aucun secret à stocker, à récupérer ou à faire tourner. Le token est émis pour le workload au besoin, il est de courte durée (en général quelques minutes), et il est lié à l'identité que la plateforme connaît déjà. La révocation est le travail de la plateforme : retirer à l'identité le droit de demander des tokens arrête les nouveaux tokens, et les tokens déjà émis peuvent être acceptés jusqu'à leur expiration. Le login dans la base de données existe toujours, reste membre de son rôle `app_*` et apparaît toujours dans le log.

Les coûts sont précis :

- **Couplage à la plateforme.** La conception fonctionne là où la plateforme la prend en charge et nulle part ailleurs, si bien qu'une équipe dont les workloads tournent sur deux plateformes construit le pattern deux fois.
- **Mapping.** Quelqu'un doit dire quelle identité de plateforme correspond à quel login et à quel rôle. Ce mapping est un grant, et il mérite la même revue que n'importe quel autre (Chapitre 14).
- **Connexions ouvertes.** Sur les moteurs courants, le token est vérifié à l'ouverture d'une connexion. Comme pour toute expiration, une connexion ouverte survit à son token (voir « Expiration et révocation »), de sorte qu'une limite de durée de vie des connexions reste nécessaire (Chapitre 5).

Le choix entre ces trois approches dépend moins de la mode que de ce que la plateforme offre et de ce que l'équipe sait exploiter.

Approche	Secret stocké?	Durée de vie	Prérequis	Coût principal
Credential statique dans un gestionnaire de secrets, rotation A/B	Oui, une paire par service	Jusqu'à la rotation suivante	Gestionnaire de secrets, job de rotation	Les consommateurs doivent relire ; le job de rotation doit continuer à tourner
Credentials dynamiques fournis par un broker	Seulement celui du broker	De quelques minutes à quelques heures, par instance	Broker, `ledgerly_access_admin`, lease sweeper	Le broker est un composant à forte valeur, toujours disponible
Tokens de workload identity	Aucun pour la base de données	Courte, en général quelques minutes	Prise en charge par la plateforme et la base de données	Couplage à la plateforme ; tokens vérifiés seulement à la connexion

Aucune de ces approches ne supprime le problème des credentials ; chacune le déplace. Le login propre du broker est un credential statique, détenu dans le gestionnaire de secrets, récupéré par la workload identity du broker et soumis à une rotation A/B. La conception par token suppose que le système d'identité de la plateforme est correct. Ce qui change, c'est le nombre de secrets, la durée pendant laquelle chacun est utile, et qui peut les lire.

TLS et vérification des certificats

Tout ce qui précède porte sur l'endroit où un credential est conservé et sur sa durée de vie. Avant d'atteindre la base de données, il traverse un réseau, et une connexion non chiffrée transporte le mot de passe, et les données, sous une forme que n'importe quoi sur le chemin peut lire. Le chiffrement en transit ferme cette porte, et c'est le minimum. Il doit tenir sur chaque tronçon. Une requête passe souvent par un pooler ou une gateway avant d'atteindre la base de données, et chacun d'eux est une connexion distincte avec son propre chiffrement et sa propre vérification de certificat. Chiffrer le tronçon du client au pooler ne dit rien du tronçon du pooler à la base de données.

Il y a une seconde tâche, plus facile à négliger : savoir que le serveur est le bon. Une connexion peut être chiffrée vers une destination qui n'est pas la base de données. La **vérification des certificats** consiste pour le client à vérifier que le certificat du serveur a été émis par une autorité en laquelle il a confiance et que ce certificat nomme l'hôte qu'il voulait atteindre. Sans elle, le chiffrement protège contre quelqu'un qui lit le trafic et ne fait rien contre quelqu'un qui répond à la connexion.

Le réglage qui compte est le mode de vérification du client. La plupart des clients de base de données proposent un mode qui chiffre sans vérifier, et il est populaire parce qu'il fonctionne avec n'importe quel certificat, y compris expiré ou auto-signé. Je considère ce mode comme la moitié du travail : il cache le trafic à un observateur passif et donne la fausse impression que le serveur est le bon. Le remède est de distribuer l'autorité de certification à chaque service et de vérifier le nom.

Cela apporte ses propres coûts. Les certificats expirent, et un certificat de serveur expiré est une panne avec une date exacte, si bien que l'expiration exige une surveillance et une procédure de renouvellement répétée. Distribuer le certificat de l'autorité à chaque consommateur représente du travail, et il faut le mettre à jour quand l'autorité change. Les certificats client, où le certificat est le credential, ne fonctionnent avec des durées de vie courtes que si quelque chose les émet automatiquement.

Quand un credential fuit

Même avec tous les contrôles ci-dessus, un credential peut fuiter, et la conception doit dire à l'avance ce qui se passe ensuite. La première étape consiste à séparer les deux questions que la panique confond : comment empêcher le credential de fonctionner, et qu'a-t-il fait pendant qu'il fonctionnait ?

Pour un credential dynamique, la première réponse est une révocation : mettre fin au lease, fermer les sessions, et le login disparaît dans le temps qu'il faut au broker pour agir. Pour un credential statique dans un dispositif A/B, une fuite suspectée n'est pas la rotation planifiée. Le job planifié attend que l'ancien login se vide, et une valeur qui a fuité reste valable tant que quelqu'un détient une session dessus. En cas de fuite, coupez d'abord le login compromis : retirez ses rôles, fermez ses sessions et chan-

gez son mot de passe sans attendre la vidange, puis publiez une nouvelle valeur sur le login inactif et basculez le pointeur actif. Les instances qui détiennent encore l'ancien login perdent leurs connexions et échouent jusqu'à ce qu'elles relisent le secret. C'est un coût en disponibilité, et je le paierais : un credential compromis qui continue de fonctionner est le problème le plus grave. Un service qui ne peut pas relire son secret a besoin d'un redémarrage, et l'équipe doit savoir de quels services il s'agit avant le jour où cela compte. Pour un mot de passe admin partagé dans l'état initial, la première réponse est la panne par laquelle ce chapitre a commencé, ce qui est l'argument le plus fort pour faire le travail avant un incident.

La seconde question trouve sa réponse dans le log. Un login qui correspond à une seule instance de workload vous permet de voir les instructions qu'il a exécutées, les tables qu'il a touchées et les heures où il était actif. Cela réduit la revue au rayon d'impact d'un seul rôle plutôt qu'à toute la base de données. La procédure pas à pas, l'ordre des actions et la manière de la répéter relèvent du Chapitre 15, qui traite de l'accès pendant les incidents. Ici, l'essentiel est qu'une équipe qui a fait de la révocation une étape de routine peut s'en servir un mauvais jour sans avoir à l'inventer.

Les régimes de protection des données et de cartes de paiement demandent en général comment l'accès est accordé, combien de temps il dure et comment il est révoqué. Les leases, les étapes de révocation et les entrées d'audit décrits ici sont les enregistrements qui répondent à ces questions.

Le plan de migration de Ledgerly

Le passage du mot de passe admin partagé aux credentials de courte durée est une suite de petites étapes réversibles. Je les ordonnerais selon le risque que chaque étape supprime et le peu qu'elle peut casser.

1. **Inventorier chaque copie du mot de passe admin.** Services, variables du pipeline, étape d'export, gestionnaire de mots de passe, connexions enregistrées dans les outils clients. La liste est le plan de travail. L'inventaire du Chapitre 1 est le point de départ.
2. **Créer un login par service, dans son rôle `app_*`.** Utilisez les rôles du Chapitre 3. À ce stade, les credentials sont statiques, mais chacun est

distinct, stocké dans le gestionnaire de secrets et lu par identité.

3. **Retirer les services du login admin un par un.** Commencez par `reporting-job`, dont le rayon d'impact est le plus petit, puis `invoice-worker`, `admin-console` et `billing-api`. Traitez `payments-service` en dernier, car c'est le seul service qui atteint `payment_tokens` et `bank_accounts`, et une erreur y coûte le plus cher. Chaque bascule est un déploiement qu'on peut annuler par un rollback.
4. **Donner aux migrations et à l'étape d'export leurs propres logins.** Le pipeline utilise un login de courte durée dans `ledgerly_migrator` (Chapitre 11) ; le rôle lui-même détient des privilèges de propriétaire permanents, et seul le credential est de courte durée. L'étape d'export utilise `app_vendor_export` (Chapitre 13). Ni l'un ni l'autre n'utilise le login admin après cette étape.
5. **Passer chaque service à la rotation A/B.** Faites en sorte que chaque service relise son secret après un échec d'authentification, exécutez la première rotation pendant une période calme et surveillez la vérification de vidange. Un service qui ne se vide pas est un constat, et ce constat est tout l'intérêt de cette étape.
6. **Introduire le broker pour les services qui se renouvellent facilement.** Commencez par un service qui redémarre souvent et détient peu de connexions. Ajoutez les autres à mesure que l'équipe apprend comment les leases se comportent. Gardez la rotation A/B statique comme fallback.
7. **Retirer le login admin partagé.** Supprimez-le de chaque configuration et du gestionnaire de mots de passe. Une fois la propriété transférée à `ledgerly_owner` (Chapitre 3), retirez-lui ses privilèges et supprimez-le, ou désactivez-le. Si l'admin se révèle être le superutilisateur du moteur, comme l'est souvent l'utilisateur master d'un service managé, il devient ce que le Chapitre 3 appelle l'instrument scellé. C'est un credential que personne n'utilise dans le travail quotidien. Il est détenu là où l'équipe plateforme peut le récupérer, et chaque récupération est enregistrée. Changez son credential une fois, après que tout l'inventaire a été traité.

Une équipe pourrait s'arrêter après l'étape cinq et être déjà bien mieux lotie : chaque service a son propre credential, le lit par une identité et en fait la rotation sans panne. Le broker est un ajout pour les équipes capables de l'exploiter.

Compromis et coûts

Chaque mesure de ce chapitre déplace un coût d'un incident vers une routine. Voici où la routine devient chère.

Dépendance au gestionnaire de secrets. Dès que les credentials viennent d'un gestionnaire de secrets, un gestionnaire de secrets injoignable empêche de nouvelles instances de démarrer et, avec des leases courts, finit par empêcher les instances en cours de renouveler. Il fait donc partie de la disponibilité du chemin vers la base de données. Exploitez-le avec le soin que vous accordez à la base de données : plus d'une instance, une restauration testée et une rotation d'astreinte qui connaît son existence. Mettre un credential en cache pour la durée d'un lease atténue une courte panne, et allonge le temps pendant lequel un credential révoqué peut encore être utilisé.

Connection storms. Si toutes les instances d'un service renouvellent à la même minute, ou redémarrent ensemble après un déploiement, la base de données subit une rafale de nouvelles connexions et authentifications. L'authentification n'est pas gratuite, et une rafale qu'elle s'inflige elle-même peut ralentir la base de données pour tous les autres rôles : un connection storm sans aucune cause extérieure. Échelonnez les renouvellements avec du jitter, gardez le pool de connexions stable entre deux renouvellements, et laissez le Chapitre 5 définir combien de connexions un rôle peut détenir.

Un débogage plus difficile. Un login qui existe pendant une heure a disparu quand quelqu'un lit le log le lendemain matin, et un nom comme `billing-api-4f2a` ne veut rien dire sans l'enregistrement du lease. Conservez les leases et leur identité aussi longtemps que le log d'audit, et donnez à la personne d'astreinte un moyen de retrouver l'instance à partir du login. Si ce moyen manque, le credential de courte durée rend l'investigation plus lente, et non plus rapide.

Un broker à forte valeur. Le login du broker peut créer des logins dans chaque rôle d'application, il vaut donc l'effort d'un attaquant. Gardez son code petit et relu, et accordez-lui l'attention que vous accordez à ce qu'il protège ; un broker qui en reçoit moins est un risque en soi.

Quand ne pas l'utiliser. Une équipe avec un seul service, une poignée d'ingénieurs et un store de secrets managé peut être bien servie par un credential statique distinct, détenu dans le gestionnaire de secrets et soumis à une rotation A/B manuelle deux fois par an. Un broker est un service de plus à exploiter et un système de plus que l'équipe doit apprendre ; en dessous d'une certaine taille, les pièces mobiles ajoutées causeront plus de pannes qu'elles n'en éviteront. Si vous ne pouvez pas dire qui recevrait un appel d'astreinte quand le broker tombe, j'attendrais. Les tokens de workload identity sont l'étape la moins chère quand la plateforme les propose, et le mapping doit tout de même être revu.

Erreurs courantes

- **Considérer le gestionnaire de secrets comme toute la réponse.** C'est tentant parce que la mesure est visible et rapide. Le mot de passe partagé reste partagé si chaque service lit le même chemin. Donnez à chaque service son propre chemin et son propre login, et revoyez qui peut lire chacun.
- **Une rotation que personne ne consomme.** Un job fait la rotation de la valeur, mais les services la lisent au démarrage et plus jamais ensuite, si bien que la nouvelle valeur reste non lue jusqu'à ce que l'ancienne soit retirée et que quelque chose échoue. Testez que les consommateurs prennent en compte une rotation, et intégrez la vérification de vidange au job.
- **Se rabattre sur une valeur intégrée à l'image.** Une copie « dernière valeur connue valide » dans l'image rend le démarrage fiable et ramène le fichier que la conception avait supprimé. Refusez le démarrage, alertez et réparez la dépendance.
- **Chiffrer sans vérifier.** Activer TLS satisfait une checklist, et le mode chiffrement seul fait fonctionner la connexion. Vérifiez le certificat et le nom d'hôte, et surveillez la date d'expiration.

- **Des durées de vie sans plafond.** Un renouvellement sans limite reconstruit un credential de longue durée. Plafonnez la durée de vie totale d'un lease et faites en sorte que l'instance se réauthentifie avec son identité.
- **Considérer le rôle du broker comme petit.** `ledgerly_access_admin` est plus large que la tâche du broker, de sorte qu'un broker compromis peut aller aussi loin que les rôles qu'il administre. Restreignez les principals autorisés à l'appeler et les rôles `app_*` que chacun peut demander, et revoyez ses entrées d'audit. Traitez ce rôle large comme un coût que vous avez accepté, et non comme une limite.

Points clés

- Chaque copie d'un credential est un lecteur, si bien que le changement le plus utile consiste à réduire le nombre de copies et à donner un nom à chaque lecture restante.
- La rotation est difficile parce qu'un credential est un état partagé ; deux logins par service permettent d'en changer un pendant que l'autre continue de fonctionner.
- Une rotation que les consommateurs ne prennent pas en compte, ou que personne n'exécute, est une entrée de calendrier et non un contrôle ; la vérification de vidange et une alerte sur un job de rotation qui a cessé de tourner font donc partie de la conception.
- Les credentials dynamiques donnent à chaque instance de workload son propre login avec une fin intégrée, au prix d'un broker, d'un enregistrement de lease et d'une dépendance au gestionnaire de secrets.
- La workload identity supprime le mot de passe stocké là où la plateforme la prend en charge, et déplace la confiance vers la plateforme et vers le mapping entre identité et login.
- C'est la vérification du certificat et du nom du serveur qui permet à TLS de protéger le credential ; le chiffrement sans vérification cache le trafic et laisse l'identité du serveur non vérifiée.
- Quand un credential fuit, la rapidité de la révocation et la clarté du log décident du degré de confinement de l'événement, et cela dépend du travail fait avant la fuite.

Les credentials décident de qui entre ; ils ne disent rien du nombre de connexions qu'ils ouvrent ni du temps pendant lequel une instruction peut occuper le primary. Le Chapitre 5 fait de la connexion elle-même une politique : des limites par rôle, du pooling, des timeouts et les labels qui permettent de repérer facilement une requête hors de contrôle.

Chapitre 5 — Les connexions comme politique

Imaginez Ledgerly par une nuit calme de mardi. Une personne de l'équipe a augmenté le parallélisme de l'agrégation nocturne de `reporting-job` pour qu'elle se termine avant le matin, un changement qui semblait anodin en revue. À 01:40, le job démarre, ouvre une connexion pour chacun de ses threads de travail et commence à lire `payments` et `invoices` sur le primary, avec le login partagé `admin`. En quelques minutes, `billing-api` commence à renvoyer des erreurs à des clients situés dans un fuseau horaire où c'est déjà le matin. Robin, d'astreinte, reçoit un appel d'astreinte pour une panne visible par les clients.

Robin ouvre la base de données et constate que rien n'est cassé. Le primary est en service, l'activité disque est normale, et aucune instruction n'a fait quoi que ce soit de travers. Le serveur dispose d'un nombre fixe de slots de connexion, le rapport en a pris la plupart, et l'API attend à la porte. Comme chaque appelant utilise le même login, la base de données ne peut pas dire que le rapport est la cause. Elle peut seulement dire que les slots sont pleins.

Ce chapitre répond à une question située entre l'identité et les données : comment faire de la connexion elle-même un endroit où une politique s'applique ? Dans le modèle en cinq parties du Chapitre 1, il agit sur deux parties. La connexion relève du Chemin, parce que le pooler décide de la manière dont les services arrivent et du nombre d'entre eux qui le peuvent. Elle est aussi la première borne de l'Exécution, parce que les limites et les timeouts disent combien un appelant, et une instruction, peuvent prendre à tous les autres. L'étape de l'exemple fil rouge est que les connexions sont façonnées par un pooler et par des limites et des timeouts par rôle.

Le problème, et pourquoi il est difficile

Une connexion est une ressource avant d'être quoi que ce soit d'autre. Sur la plupart des moteurs, chacune coûte au serveur de la mémoire et une part de son ordonnancement, et le serveur n'en autorise qu'un nombre fixe. La limite est une propriété du serveur, pas de l'appelant, si bien que le premier appelant qui en demande beaucoup en obtient beaucoup. Rien dans la configuration par défaut ne dit que le rapport nocturne devrait avoir moins de slots que l'API destinée aux clients.

Un mot de passe ne touche à rien de tout cela. Il décide de qui entre ; il ne dit rien du nombre de slots détenus ni de leur durée. C'est pourquoi un livre qui traite d'autre chose que de la distribution de credentials a besoin d'un chapitre sur les connexions.

Trois habitudes aggravent le problème. Premièrement, le nombre de connexions est un produit, pas une somme : les instances d'un service multipliées par la taille du pool de chaque instance, multipliées encore chaque fois qu'un autoscaler ajoute une instance. Supposons qu'une équipe fixe un pool de vingt sur un service qui monte à dix instances. Elle a demandé deux cents connexions, et ne l'a probablement pas décidé délibérément. Deuxièmement, une connexion est détenue aussi longtemps que son instruction s'exécute, si bien qu'une requête lente occupe un slot tout ce temps, et les slots sont ce dont l'appelant suivant a besoin. Troisièmement, un appelant en échec a tendance à insister. Un client qui atteint son timeout et fait un retry immédiatement transforme une base de données lente en foule à la porte.

La réponse évidente est d'augmenter la limite du serveur. Cela aide pendant une semaine, puis rend l'incident suivant plus grave. Plus de connexions signifie plus de travail simultané sur le même primary, et la base de données n'exécute pas plus vite cent instructions à la fois parce que cent sont autorisées.

Les contrôles vivent dans plusieurs endroits dont personne n'est responsable dans leur ensemble : la définition du rôle, le pool de l'application, un éventuel pooler, et les réglages qui décident combien de temps une instruction peut s'exécuter. La disponibilité est l'une des propriétés en jeu dans le modèle de menaces du Chapitre 2, et les accidents y dominent. Un login partagé transforme chaque accident en accident de tout le monde.

Dans l'état initial, donc, les cinq services et chaque ingénieur sur un VPN puisent dans un même pool de slots du serveur sous un même nom. Il n'y a pas d'autre timeout que les valeurs par défaut de chaque bibliothèque cliente, et aucun enregistrement ne dit quel appelant détient quelle connexion. Les sections qui suivent remplacent cela par un budget par rôle, un pooler qui l'applique, des timeouts qui fixent une limite haute à la durée pendant laquelle quoi que ce soit peut détenir un slot, et une étiquette sur chaque connexion qui dit à qui elle appartient.

Un budget de connexions par rôle

La première étape est la moins chère. Le Chapitre 3 a donné à chaque service son propre rôle applicatif, ce qui signifie qu'une limite peut enfin s'attacher à un nom. Une limite par rôle dit : ce principal peut détenir au plus ce nombre de connexions, quel que soit le nombre d'instances qu'il exécute ou l'insistance avec laquelle il demande. Quand il atteint la limite, les nouvelles connexions de ce rôle sont refusées, et tous les autres rôles continuent de fonctionner.

Ce refus est la politique. Il fait passer la défaillance de « la base de données est pleine pour tout le monde » à « cet appelant n'a plus de slots », et le second est un bien meilleur appel d'astreinte à recevoir pour Robin. Le rayon d'impact d'un appelant gourmand devient l'appelant lui-même, et le refus nomme l'appelant, ce qui constitue la moitié imputabilité de la même étape.

Les chiffres demandent un raisonnement, pas une formule. Je partirais de la limite du serveur vers le bas. D'abord, soustrayez une réserve que rien d'autre ne peut utiliser, parce que les personnes et les outils qui réparent le système ont aussi besoin de se connecter. La réserve couvre les sessions de la plateforme et les logins des trois principals qui utilisent `ledgerly_access_admin` (l'étape du pipeline qui applique les définitions de rôles, le credential broker et la gateway d'accès). Répartissez ensuite le reste entre les rôles applicatifs selon ce dont chacun a besoin au plus fort de son activité, plus une marge. Le tableau suivant est une illustration pour le primary. Les chiffres sont hypothétiques et ne constituent pas une recommandation.

Principal	Rôle	Limite de connexions (illustrative)	Pourquoi cette taille
`billing-api`	`app_billing_api`	80	Service le plus chargé ; sert l'application web et l'API publique
`invoice-worker`	`app_invoice_worker`	30	Piloté par une file ; la concurrence est fixée par le nombre de workers
`payments-service`	`app_payments_service`	30	Régulier ; dialogue avec des fournisseurs externes entre deux instructions
`admin-console`	`app_admin_console`	15	Actions étroites et auditées, effectuées par quelques humains
Migrations	`ledgerly_migrator`	3	Une migration à la fois, depuis le pipeline
Sessions humaines sur le primary	Logins de session de type JIT et break-glass, créés par la gateway	8	Seules les sessions avec une demande approuvée s'exécutent ici (Chapitre 6) ; les lectures courantes vont vers `ledgerly_replica_2`. Les rôles `human_jit_write` et `human_breakglass` ne portent aucune limite, car aucun login n'est membre du premier et le second est accordé par session
`reporting-job`	`app_reporting_job`	10	Transition seulement : il lit le primary dans l'état initial et pendant le déplacement, et la ligne passe à zéro dans l'état cible
Réserve	administration des accès et plateforme	10	Uniquement pour la réparation et l'émission de logins
Non attribué	aucun	14	Marge, pour qu'une erreur d'arrondi ne devienne pas une panne

Supposons que le serveur autorise 200 connexions. Les lignes attribuées totalisent 186, et les 14 restantes sont délibérément laissées sans affectation. Les sessions humaines courantes ne figurent pas dans ce tableau parce qu'elles ne s'exécutent pas sur le primary. Leur budget est fixé sur `ledgerly_replica_2`, où vont les lectures courantes ; disons 20 slots là-

bas. La ligne `reporting-job` n'existe que tant que le rapport lit encore le primary. Dans l'état cible, il lit depuis `ledgerly_replica_2`, avec son propre budget sur cet hôte, fixé au niveau de la couche de connexion dans le pooler (le Chapitre 6 traite du déplacement). Les définitions de rôles pourraient porter les limites dans une notation comme celle-ci, qui est illustrative et n'appartient à aucun moteur :

```
// illustrative notation: the limit belongs to the role, not to a
person
ROLE LIMIT app_billing_api      CONNECTIONS 80;
ROLE LIMIT app_invoice_worker   CONNECTIONS 30;
ROLE LIMIT app_payments_service CONNECTIONS 30;
ROLE LIMIT app_admin_console    CONNECTIONS 15;
ROLE LIMIT ledgerly_migrator    CONNECTIONS 3;
ROLE LIMIT app_reporting_job    CONNECTIONS 10;
```

> **Remarque :** Les moteurs diffèrent quant à ce à quoi une limite ou un réglage s'attache. Certains comptent les connexions par rôle, d'autres par login, et d'autres seulement par base de données ou par serveur. Sur certains moteurs, une limite ou une valeur par défaut écrite sur un rôle ne s'applique qu'aux connexions qui se connectent avec le rôle lui-même, et n'est pas héritée par les logins qui en sont membres. Le Chapitre 4 donne à chaque instance de service son propre login dynamique, membre du rôle `app_*`, si bien que sur ces moteurs la limite au niveau du rôle et les timeouts au niveau du rôle ci-dessous ne s'appliqueraient pas, sans que rien ne le signale.

Deux endroits peuvent les porter à la place : le broker, qui peut fixer une limite par login et les mêmes réglages lorsqu'il crée chaque login, et le pooler, qui peut porter les réglages mais peut ne pas être capable d'additionner les logins d'un rôle en un seul total. La section sur le pooler ci-dessous présente trois façons d'obtenir un total par rôle quand il ne le peut pas.

Deux propriétés rendent le tableau utile. Les limites sont vérifiées là où la connexion commence, de sorte qu'un appelant apprend le refus immédiatement et non après la première instruction lente. Et la réserve se trouve en dehors de tout rôle applicatif, de sorte que, la pire nuit, l'équipe plate-

forme peut encore se connecter pour réparer. Un budget qui remplit tout le serveur a le même défaut qu'un mot de passe admin partagé : il fonctionne jusqu'au moment où quelqu'un en a le plus besoin.

Les sommes n'ont pas à être prudentes. Si `billing-api` et `payments-service` atteignent rarement leur pic en même temps, vous pouvez laisser les limites applicatives totaliser plus que ce que le serveur autorise, et accepter que deux pics simultanés puissent entrer en collision. Je ne le ferais qu'en conservant la réserve, et seulement après que l'équipe a observé l'utilisation réelle pendant un certain temps. Le choix par défaut est un budget dont la somme tombe juste, parce qu'un budget qui ne peut pas être dépassé est plus facile à raisonner à trois heures du matin.

Le pooling: là où les connexions sont partagées

Un pool conserve un ensemble de connexions ouvertes et les prête aux appelants, de sorte que chaque requête ne paie pas une nouvelle connexion et que la base de données voit un nombre borné de connexions. Il y a deux endroits où en placer un.

Presque chaque service a déjà un pool dans sa bibliothèque de base de données, et il doit être dimensionné selon le travail qu'une instance peut faire à la fois, pas selon le trafic qu'elle espère servir. Un petit pool qui fait attendre brièvement les appelants est plus clément pour la base de données qu'un grand, et il doit abandonner après une courte attente avec une erreur que le service peut transformer en réponse claire. Commencez petit, observez combien de temps les appelants attendent, et augmentez la taille sur la base de preuves.

Le pooler devant la base de données

Un **pooler** est un proxy de base de données dont la tâche principale est d'accepter de nombreuses connexions clientes et de les multiplexer sur un nombre plus petit de connexions serveur. PgBouncer et ProxySQL en sont des exemples courants. Le service se connecte au pooler comme s'il s'agissait de la base de données, et le pooler décide quand une vraie connexion est prêtée.

La question que le livre ne cesse de poser s'applique ici aussi : quel nom la base de données voit-elle ? Le Chapitre 4 a donné à chaque instance de service son propre login dynamique, créé par le credential broker et membre du rôle `app_*` du service. Le pooler doit s'inscrire dans ce modèle, et je le construirais sous cette forme. L'instance présente son propre login et son propre credential au pooler. Le pooler les vérifie auprès de la base de données et conserve des connexions serveur sous ce même login. Il ne détient pas de stockage de mots de passe propre, et il ne détient pas non plus de login propre que tous les clients partageraient.

Un credential qui a subi une rotation ou qui a expiré atteint le pooler par la base de données, pas par le pooler. Quand un lease prend fin ou est révoqué, le broker supprime le login (Chapitre 4), si bien que la tentative suivante du pooler de s'authentifier ou d'ouvrir une connexion serveur sous ce login est refusée, et il ferme les connexions serveur inactives qu'il détient pour ce login.

La manière dont une limite par rôle tient encore n'a pas de réponse universelle, et je ne supposerais pas que votre pooler peut en donner une. Beaucoup de poolers gardent leurs limites par paire login et base de données, et certains ajoutent une limite par login ou par base de données. Je ne connais aucune capacité générale qui additionne les connexions de nombreux logins différents en un seul plafond au niveau du rôle ; lisez donc la documentation de votre pooler au lieu de croire ce chapitre sur parole. Si le vôtre ne le peut pas, le total par rôle doit venir d'ailleurs. Il y a trois endroits, et chacun a un prix.

- **Le pooler se connecte en upstream avec un login stable par rôle.** Le credential dynamique de l'instance authentifie alors le client auprès du pooler seulement, et la limite par login du pooler est exactement la limite du rôle. Le prix est le modèle d'identité du Chapitre 3 :
  - Le log de la base de données montre un seul nom par rôle, si bien que l'attribution à une instance passe au log du pooler et à l'étiquette applicative décrite plus bas.
  - Le pooler doit vérifier lui-même les credentials dynamiques, ce qui fait de lui un consommateur de l'état des leases du broker.
  - Il détient aussi un credential de longue durée pour chaque rôle, à protéger et à soumettre à une rotation A/B comme le décrit le Chapitre 4.

    - Révoquer le login d'une instance au niveau du broker ne ferme plus ses connexions serveur ; le pooler doit mettre fin aux sessions de ce client.

- **La base de données applique une limite par rôle, là où le moteur la prend en charge pour les logins membres.** Elle est exacte, elle se trouve là où rien ne peut la contourner, et les logins des instances ainsi que leur attribution restent intacts. Le prix est que la prise en charge varie, comme le dit la remarque ci-dessus. Là où cela fonctionne, le refus vient de la base de données après que la connexion a déjà coûté une authentification, et rien n'est multiplexé.
- **Le broker plafonne le nombre de logins qu'il émet par rôle.** Avec une limite par login fixée à la création de chaque login, le total du rôle ne peut pas dépasser le nombre de logins multiplié par cette limite. Le prix est qu'il s'agit d'une borne supérieure et non d'un décompte exact, et que le plafond doit correspondre au plafond d'instances de l'autoscaler. Une montée en charge au-delà du plafond devient un déploiement qui ne peut pas obtenir de credentials, ce qui est un problème de disponibilité, et les renouvellements qui font brièvement se chevaucher un ancien et un nouveau login ont besoin de place sous le plafond.

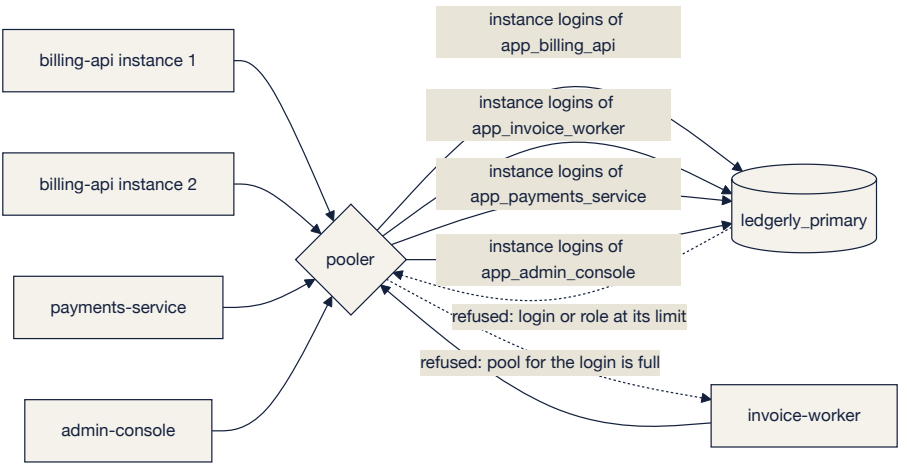
J'essaierais d'abord la limite côté base de données, là où le moteur la respecte pour les logins membres. Là où ce n'est pas le cas, le plafond du broker préserve le modèle d'identité et coûte un plafond à maintenir. Je ne prendrais le login upstream unique par rôle que lorsque le nombre de connexions impose de multiplexer entre instances, et je consignerais par écrit ce que le log ne dit plus. Les trois valent mieux que la conception qui fait correspondre chaque client de chaque rôle à un seul login partagé. Elle recrée le compte admin partagé une couche plus bas : le log montre de nouveau un seul nom, et une limite s'applique à tout le monde à la fois.

Dans la forme pass-through recommandée, le pooler apporte peu de multiplexage. En mode transaction, les clients d'une instance partagent les connexions serveur de cette instance, mais deux instances ne partagent pas entre elles, si bien que le total d'un rôle est au moins le nombre d'instances vivantes multiplié par un petit minimum. Ce que le pooler apporte, ce sont trois autres choses.

- **Un refus central.** Un seul endroit dit non, par login et par pool, de la même façon pour chaque bibliothèque cliente.
- **Les mêmes limites pour chaque service.** Les tailles de pool et les attentes de chaque service cessent d'être la seule ligne de défense.
- **Un chokepoint réseau.** Les services n'entrent dans la base de données que par le pooler. La network policy, qui dit quels hôtes peuvent ouvrir une connexion TCP, que le Chapitre 6 définit et que le Chapitre 7 écrit, admet les hôtes du pooler et personne d'autre, de sorte qu'un credential qui fuit depuis un service est inutilisable depuis tout endroit d'où le pooler n'est pas joignable. La portée borne une fuite, comme le dit le Chapitre 4, et le pooler est l'endroit où cette borne est facile à énoncer.

Le coût est un nouveau système sur le chemin, traité dans les compromis plus bas. Pour un seul petit service avec une seule instance, le pool de l'application et une limite de rôle suffisent, et le pooler ne vaut pas d'être ajouté tant que le nombre d'instances ou le besoin d'un chokepoint unique ne dit pas le contraire.

Le diagramme suivant montre la forme pass-through, qui laisse les logins des instances visibles pour la base de données. Repérez les deux endroits où un appelant est refusé : au pooler, quand le pool d'un login est plein, et à la base de données, quand un login ou, là où le moteur le prend en charge, le rôle atteint sa limite.



Mode session et mode transaction

Un pooler prête une connexion serveur pour une période, et la durée de cette période est son **mode**. En mode session, le client garde la même connexion serveur de la connexion à la déconnexion ; le pooler n'économise rien d'autre que le coût de connexion. En mode transaction, le client ne détient une connexion serveur que pendant la durée d'une transaction, et la transaction suivante peut s'exécuter sur une autre. Le mode transaction multiplexe bien plus de clients sur peu de connexions, et c'est pourquoi il est attrayant.

Il s'accompagne d'un coût facile à manquer, parce qu'il concerne l'autorisation et non la performance. Tout ce qu'un client règle sur sa session vit sur une connexion serveur, et en mode transaction le client n'en possède aucune. Des réglages faits en dehors d'une transaction peuvent disparaître à l'instruction suivante, ou pire, rester sur une connexion que le pooler prête à quelqu'un d'autre. Les éléments qui dépendent de l'état de session incluent les tables temporaires, les instructions préparées dans de nombreuses configurations, les verrous consultatifs, les timeouts au niveau de la session, et toute valeur de contexte que votre propre conception stocke dans la session.

Ce dernier élément touche directement le livre. Au Chapitre 3, une règle de ligne peut dépendre d'une valeur comme `current_case_customer()`, et `app_admin_console` transporte l'identité d'un principal humain jusqu'au log d'audit et jusqu'à `audit_events`. Si l'une de ces valeurs est réglée une fois par session et lue plus tard, alors en mode transaction elle peut manquer ou, dans le mauvais cas, appartenir au client précédent.

> **Avertissement :** En mode transaction, une valeur de contexte réglée au niveau de la session peut rester sur une connexion serveur et être lue par la transaction du client suivant. Une règle de ligne ou un enregistrement d'audit qui la lit peut alors agir pour le mauvais client ou la mauvaise personne. Réglez ces valeurs à l'intérieur de la transaction, avec une portée limitée à celle-ci, et faites échouer le code qui les lit quand la valeur est absente.

Le pattern qui tient est de régler le contexte à l'intérieur de la transaction et de le laisser se terminer avec elle. Le bloc suivant en montre la forme, et il est écrit pour rester correct quand le travail échoue en cours de route.

```
function runAsPrincipal(principal, customerId, work):
    transaction:
        // transaction-scoped: it disappears at commit or rollback, so
        // the next client cannot inherit it from this connection
        setLocalContext("acting_principal", principal.id)
        setLocalContext("case_customer", customerId)
        try:
            work()
        on ContextMissing:
            // a row rule or audit write found no value: stop, do not
guess
            raise ActionRefused
        // the commit happens here, once, after all of the work
succeeded
    // on any error the transaction rolls back, and the context goes
with it
```

Le log de la base de données montre toujours le login de l'instance, mais en mode transaction les clients d'une instance partagent ses connexions serveur, si bien que le log ne peut pas dire lequel des threads ou des requêtes de l'instance détenait un slot. Pour le savoir, il vous faut la vue propre du pooler sur ses clients, et elle doit être prévue avant le premier incident. C'est pourquoi les sections suivantes sur les timeouts et le nommage comptent.

Les humains n'utilisent pas ce pooler. Ils atteignent la base de données par la gateway d'accès, qui crée un login pour chaque session (Chapitre 7). Leurs limites sont le sujet de la dernière partie de la section suivante, parce qu'une personne peut faire une chose qu'un service ne peut pas faire : lever la limite.

Les timeouts comme politique

Un slot est prêté aussi longtemps qu'une instruction ou une transaction s'exécute, et rien, sur la plupart des moteurs, ne met fin à l'une ou à l'autre par défaut. Les timeouts sont la façon dont l'équipe plateforme dit combien de temps un rôle peut détenir une ressource, et ils sont la partie de ce chapitre qui borne l'Exécution : une instruction qui s'exécute trop longtemps est annulée là où elle s'exécute, et le slot est rendu. Ils relèvent de

la définition du rôle et de la configuration du pool, et non de la bonne volonté de chaque bibliothèque cliente, et c'est pourquoi ce chapitre les traite comme une politique.

Trois timeouts relèvent de la base de données et deux du pool.

Timeout	Relève de	Ce qu'il arrête	Qui le ressent
Instruction	Base de données	Une requête qui s'exécute bien plus longtemps que ce dont le rôle devrait jamais avoir besoin	L'appelant de cette instruction
Attente de verrou	Base de données	Une instruction en file derrière un verrou, et tous ceux en file derrière elle	L'appelant ; ainsi que tout appelant qui attend sur la même ligne
Inactivité en transaction	Base de données	Une transaction ouverte puis abandonnée, qui détient des verrous et des slots	L'application qui l'a ouverte
Durée de vie de la connexion	Pool	Une connexion qui reste ouverte pendant des jours et garde son ancien credential ou son ancien état	Le pool, qui doit se reconnecter
Attente d'acquisition dans le pool	Pool	Un appelant qui attend indéfiniment une connexion libre	Le service qui traite la requête

Des rôles différents ont besoin de valeurs différentes, parce qu'ils font un travail différent. Une requête dans `billing-api` est une personne qui attend une page ; une instruction qui a besoin de trente secondes à cet endroit est un bug. `invoice-worker` peut légitimement exécuter une instruction plus longue pendant qu'il construit un lot. `reporting-job` exécute de longues lectures par conception, sur une réplique où une longue lecture ne bloque pas les écritures. Une valeur par défaut partagée serait mauvaise pour tous, si bien que les valeurs s'attachent au rôle. Le bloc ci-dessous utilise une notation illustrative, et les valeurs sont hypothétiques.

```
// illustrative notation: defaults for the role; see the note above on
// members, because the broker or the pooler may have to apply them
per login
ROLE SETTING app_billing_api      statement_timeout 5s,  lock_wait 2s,
idle_in_transaction 10s;
ROLE SETTING app_invoice_worker   statement_timeout 30s, lock_wait 5s,
idle_in_transaction 30s;
ROLE SETTING app_reporting_job    statement_timeout 15m, lock_wait 5s,
idle_in_transaction 60s;
ROLE SETTING human_readonly       statement_timeout 60s, lock_wait 5s,
idle_in_transaction 5m;
ROLE SETTING ledgerly_migrator    statement_timeout none, lock_wait
3s, idle_in_transaction 60s;
```

La dernière ligne reflète une leçon de migration que traite le Chapitre 11 : un changement de schéma peut s'exécuter longtemps, mais il ne doit attendre qu'un bref instant un verrou.

Deux règles d'ordre comptent davantage que les valeurs individuelles. Le timeout d'instruction de la base de données doit être plus court que la deadline de requête de l'application, car si l'application abandonne la première, elle laisse derrière elle une instruction qui s'exécute encore et le slot reste occupé ; et l'attente d'acquisition du pool doit être la plus courte de toutes, pour que les appelants apprennent tôt la congestion. Quand une deadline se déclenche quand même en premier, la connexion peut encore porter une instruction en cours, si bien que le service la ferme et ne la prête pas à l'appelant suivant.

La transaction inactive est le cas surprenant. Un service qui ouvre une transaction, appelle un fournisseur de paiement externe et attend avec la transaction toujours ouverte laisse la lenteur du fournisseur détenir des verrous et un slot sur la base de données. La correction à long terme est de garder les appels externes hors des transactions, ce qui est un changement de code ; le timeout garde le coût borné pendant que ce changement attend dans le backlog.

Une connexion qui vit des semaines garde tout ce avec quoi elle est née, si bien qu'une durée de vie maximale fait remplacer régulièrement ses connexions par le pool. Elle doit être plus courte que la durée de vie du credential, avec une certaine variation d'une connexion à l'autre pour

qu'elles ne se reconnectent pas toutes à la même minute. Le Chapitre 4 explique pourquoi un credential expiré ne ferme pas une session déjà ouverte ; la durée de vie est la façon dont le pool ferme les siennes.

Des valeurs par défaut qu'une session peut lever

Un timeout fixé comme valeur par défaut d'un rôle est une valeur par défaut. Sur beaucoup de moteurs, une session peut modifier ses propres réglages, y compris son timeout d'instruction et son mode lecture seule, et peut changer de rôle. Pour un service, c'est acceptable : les limites existent pour les accidents, et le code ne lève pas son propre timeout par accident. Pour une personne munie d'un client SQL, ce ne l'est pas, parce que la session qui a besoin de la limite est la session qui peut la lever.

Deux contrôles se trouvent là où la session ne peut pas les atteindre. La gateway ouvre chaque session humaine avec ses limites appliquées et refuse une demande de modification des réglages figurant sur une liste protégée ; le Chapitre 9 explique comment la gateway procède. Le coût est que les clients qui envoient de tels réglages à la connexion cessent de fonctionner jusqu'à ce que quelqu'un les ait listés. Les sessions humaines tiennent leurs limites de la gateway : elle fixe une limite de connexions par login selon le type de session lorsqu'elle crée le login de session, et elle compte combien de sessions une personne peut détenir. Ni l'une ni l'autre n'est une limite de rôle, puisque `human_jit_write` est un plafond dont aucun login n'est membre. Ni l'une ni l'autre n'est laissée au client. Pour toute limite qui n'a ni l'une ni l'autre, le tableau des contrôles doit dire qu'il s'agit d'une valeur par défaut et non d'une garantie.

Nommer l'appelant

Quand Robin regarde les connexions, la question utile est « à qui est-ce ? », et le login répond à la première moitié. Un rôle dit qu'une connexion appartient à `billing-api`. Il ne dit pas quelle instance, quelle release ni quelle exécution d'un job planifié. La plupart des moteurs et des poolers permettent à un client d'attacher une courte étiquette à sa connexion, généralement appelée application name, et elle apparaît dans le log de la base de données et dans la vue de l'activité en cours.

Une convention vaut plus que la fonctionnalité. Je mettrais trois choses dans l'étiquette : le service, l'instance ou l'exécution, et la release. Par exemple, `reporting-job/run-0412/v31` dit quel job, quelle exécution nocturne et quel build. L'étiquette transforme « quelqu'un détient quarante connexions » en « l'exécution du mardi du rapport détient quarante connexions, et c'est le nouveau build ».

Deux mises en garde l'accompagnent. Premièrement, l'étiquette est fournie par le client et peut prendre n'importe quelle valeur ; elle sert donc au diagnostic et à l'attribution, pas à l'autorisation. Le login est l'identité ; l'étiquette est une note sur l'enveloppe. Deuxièmement, un pooler en mode transaction partage les connexions serveur entre clients, si bien que l'étiquette côté serveur peut ne pas être celle du client. Certains poolers la transmettent pour chaque transaction, d'autres non, et le log propre du pooler sur ses clients est alors l'enregistrement qui porte le nom. Vérifiez ce que fait votre configuration avant d'en dépendre.

L'étiquette alimente aussi le Chapitre 14 : un log d'audit est plus facile à lire quand chaque ligne porte le service et l'exécution, pas seulement un login. Le coût est une ligne dans le démarrage de chaque service, et une convention de plus à maintenir quand un nouveau service apparaît.

Les connexions ne sont pas la seule ressource qu'un rôle peut consommer. La mémoire pour les tris et les jointures, l'espace disque temporaire, le CPU et le nombre de workers parallèles sont aussi partagés, et les moteurs diffèrent largement quant à leur capacité à les limiter par rôle. Là où le moteur ne le peut pas, l'isolation doit venir du placement : un gros lecteur va sur une réplique, ce qui est l'argument du Chapitre 6. Vérifiez quelles limites votre moteur peut fixer par rôle avant d'en promettre une à qui que ce soit.

Ledgerly: la nuit où le rapport a dévoré l'API

Revenons à ce mardi soir. L'état initial l'explique en une phrase : `reporting-job` s'exécutait en tant qu'`admin` sur le primary, si bien qu'il était en concurrence pour les mêmes slots que `billing-api` sous le même nom, sans limite pour l'un ni l'autre et sans timeout sur quoi que ce soit.

Le plan d'Alex ne commence pas par le pooler. Il commence par ce qui coûte le moins et protège le plus, et descend la liste à mesure que l'équipe apprend.

1. **Des limites par rôle.** Avec les rôles applicatifs du Chapitre 3 en place, chacun reçoit une limite de connexions issue d'un tableau comme celui ci-dessus. Le rapport perd la capacité de prendre d'un coup les slots de l'API, même s'il lit encore le primary la première nuit.
2. **Des timeouts par rôle.** `app_billing_api` reçoit un timeout d'instruction court et une limite d'inactivité en transaction courte, et le rapport reçoit un timeout d'instruction long qui ne s'applique pas à l'API.
3. **Des étiquettes.** Chaque service et chaque exécution planifiée règle son étiquette. Le prochain appel d'astreinte nomme l'exécution, pas seulement le rôle.
4. **Un pooler.** Les cinq services se connectent par lui et il devient leur seule entrée, avec le total par rôle tiré de l'un des trois endroits ci-dessus.
5. **Un déplacement vers la réplique.** `reporting-job` lit depuis `ledgerly_replica_2` avec son propre budget, et le primary ne porte plus son travail. Le Chapitre 6 décrit le déplacement lui-même.
6. **Une alerte avant le refus.** L'équipe plateforme surveille la saturation de chaque pool, pas seulement les refus. Un pool qui reste, disons, à quatre-vingt-dix pour cent de sa limite pendant des heures dit « augmentez la limite ou corrigez l'appelant » tant qu'il est encore temps.

Le job a aussi besoin d'un changement qui lui est propre, parce qu'une limite qui refuse le job n'est utile que si le job réagit bien au refus. Un job qui fait un retry immédiatement après un refus est la foule à la porte en plus petit.

```
function acquireWithBackoff(role):
    for attempt in 1..maxAttempts:
        try: return acquireConnection(role, waitUpTo: acquireWait)
        on PoolEmpty: sleep(backoffWithJitter(attempt))  // slow down,
do not retry harder
    raise ServiceBusy
```

Avec ces mesures en place, le même changement qui a augmenté le parallélisme du rapport aurait un résultat différent. Le job atteint sa limite et est refusé, il ralentit de lui-même et prend plus de temps, et l'API ne manque pas de slots à cause de lui. Un job qui abandonne un lot doit aussi laisser un enregistrement de ce qu'il a sauté, pour que personne ne lise des agrégats à moitié construits comme s'ils étaient complets ; cela relève de la conception propre du job et dépasse ce chapitre. La revue qui a approuvé le changement aurait encore été correcte, parce que la conception ne compte plus sur un relecteur pour remarquer l'interaction entre deux services.

Compromis et coûts

Les limites par rôle coûtent du réglage. Une limite trop basse refuse une charge légitime, et le refus ressemble à une panne pour le service qui le reçoit. Les limites ont besoin d'un responsable et d'une revue chaque fois qu'un service monte en charge ou qu'un nouveau workload apparaît, parce que la somme des instances multipliées par les tailles de pool peut dépasser la limite sans qu'aucun changement isolé ne paraisse faux. L'équipe plateforme devient l'endroit où ces questions sont posées, ce qui est utile et constitue aussi une nouvelle file d'attente.

Un pooler est un nouveau système sur le chemin. Chaque requête passe désormais par lui, et quand il tombe, chaque service tombe. Il doit être disponible pendant l'incident où vous en avez le plus besoin, il lui faut donc sa propre redondance, sa propre surveillance et son propre runbook. En mode transaction, il supprime aussi des fonctionnalités de session dont dépend une partie du code, et il rend le log de la base de données plus difficile à lire. Si vous avez un seul petit service avec une seule instance et un pool de dix, le pool propre de l'application plus une limite de rôle suffisent.

Les timeouts annulent du travail légitime. Un timeout d'instruction trop serré tue une bonne instruction, et un retry naïf répète le travail et l'annulation. Les services doivent gérer l'annulation, et pour les écritures cela signifie se demander si l'opération peut être exécutée deux fois sans dommage. Les valeurs pour `ledgerly_migrator` et pour les corrections de données (Chapitre 10) diffèrent de celles des applications, ce qui fait davantage de configuration à garder correcte.

Une limite que personne ne surveille ne protège rien. Un timeout d'instruction si long qu'il ne se déclenche jamais, ou une limite de rôle qu'aucune alerte ne couvre, ajoute une ligne à la configuration et aucune borne. Le test est de savoir si quelqu'un remarquerait que le contrôle a cessé de fonctionner.

Les étiquettes sont bon marché et falsifiables. Elles aident les personnes et coûtent de la discipline, mais elles ne prouvent rien, et un log qui s'appuie sur elles pour l'attribution s'appuie sur une convention.

Quand ne pas utiliser ces contrôles : n'ajoutez pas de limites par rôle avant que les rôles existent, car une limite sur le rôle partagé `admin` ne fait que limiter tout le monde à la fois. Ne fixez pas de timeouts agressifs sur un service avant de savoir combien de temps durent ses instructions normales, parce que la première exécution de facturation annulée apprendra à l'équipe à désactiver les timeouts. Mesurez d'abord la normale, puis fixez la limite au-dessus.

Erreurs courantes

- **Augmenter la limite du serveur pour régler un incident.** C'est tentant parce que cela fonctionne immédiatement et ne coûte rien aujourd'hui. Cela déplace la défaillance vers une foule plus grande et supprime le signal précoce. Donnez à chaque rôle un budget et laissez la limite du serveur égale à la somme.
- **Dimensionner le pool pour le trafic que vous espérez avoir.** Plus de connexions donnent l'impression de plus de capacité, mais la base de données a une quantité fixe de concurrence utile. Dimensionnez le pool pour le travail qu'une instance peut faire à la fois.
- **Faire correspondre chaque client à un seul login dans le pooler.** C'est la configuration facile, et elle ramène le compte admin partagé. Gardez le login propre de chaque instance, ou choisissez explicitement un login stable par rôle et acceptez ce que le log ne dit plus.
- **Utiliser le mode transaction avec un état de session.** Le code fonctionnait en mode session, si bien que personne ne le teste en mode transaction. Le contexte des règles de ligne, l'identité pour l'audit et les tables temporaires doivent vivre à l'intérieur de la transaction, ou le mode doit rester au niveau session pour ce rôle.

- **Un seul jeu de timeouts pour tous les rôles.** Une valeur par défaut unique est trop serrée pour le traitement par lots et trop lâche pour l'API, et l'issue habituelle est qu'elle est relevée pour tout le monde. Attachez les valeurs au rôle.
- **Traiter l'application name comme une identité.** L'étiquette est fixée par le client. Utilisez le login pour authentifier et l'étiquette pour expliquer.
- **Remplir le serveur avec les budgets applicatifs.** Quand les limites totalisent tout le serveur, l'équipe plateforme ne peut pas se connecter pendant l'incident. Gardez une réserve en dehors de tout rôle applicatif.

Points clés

- Une connexion est une ressource limitée et un point de politique : une limite qui appartient au serveur et non à un rôle laisse le premier appelant gourmand prendre la part de tout le monde.
- Un budget de connexions par rôle transforme « la base de données est pleine pour tout le monde » en « cet appelant n'a plus de slots », ce qui réduit le rayon d'impact d'un job hors de contrôle au job lui-même.
- Gardez une réserve de connexions en dehors de tout rôle applicatif, pour que les personnes et les outils qui réparent le système puissent toujours se connecter.
- Le pooler est l'endroit par lequel les services entrent, c'est donc la place du refus central et du chokepoint réseau. Beaucoup de poolers ne savent pas additionner les logins dynamiques d'un rôle, si bien que le total par rôle vient de la base de données, du broker ou d'un login upstream par rôle, chacun avec un prix nommé, comme l'expose la section sur le pooler.
- Le pooling en mode transaction supprime l'état de session, si bien que tout contexte dont dépend une règle de ligne ou un enregistrement d'audit doit être réglé à l'intérieur de la transaction et doit échouer bruyamment quand il manque.
- Les timeouts d'instruction, d'attente de verrou et d'inactivité en transaction appartiennent au rôle, et le timeout d'instruction de la base de données doit se déclencher avant la deadline de l'application. Une ses-

sion peut lever une valeur par défaut ; pour les personnes, la limite doit donc être appliquée là où elles ne peuvent pas la modifier.

- Les application names rendent les connexions explicables mais ne prouvent rien ; le login reste donc l'identité et l'étiquette reste une note.

Une limite de connexions et un pooler décident du nombre d'appelants qui atteignent le primary, mais ils ne changent pas la quantité de travail qui y arrive. Le Chapitre 6 franchit l'étape suivante : déplacer les lectures vers des répliques et d'autres copies, et les questions d'accès qui arrivent avec chaque copie.

Chapitre 6 — Lectures, répliques et copies de données

Imaginez un mardi matin chez Ledgerly, quelques jours avant la fin du mois. On a demandé un chiffre à Jordan, de l'équipe data : combien de factures ont été émises le mois dernier, réparties par segment de clientèle et selon qu'une tentative de paiement avait déjà échoué ou non. La copie d'analytics serait l'endroit naturel où poser la question, mais c'est une restauration nocturne, et cette restauration date d'hier. La question exige les données du jour. Jordan ouvre un client de bureau, se connecte avec le login admin partagé et exécute une requête qui joint `invoices`, `invoice_lines` et `payments` sur tout le mois.

Rien n'est faux dans la requête. C'est une question raisonnable, écrite par une personne compétente. Mais la connexion pointe vers le primary, parce que c'est l'hôte que le client a mémorisé, et la requête lit plusieurs grandes tables pendant que `billing-api` sert des clients sur le point de payer leurs factures. L'application web ralentit, l'astreinte se déclenche pour Robin, et Robin et Jordan passent vingt minutes à comprendre que le rapport et le ralentissement sont un seul et même problème. Personne n'a rien fait d'inhabituel. La conception offrait un seul endroit où lire, et chaque lecture y allait.

Ce chapitre ajoute une étape de l'état cible : les lectures vont vers les répliques, et le reporting ne touche jamais le primary. Dans le modèle en cinq parties du Chapitre 1, il s'agit du Grant et du Chemin qui travaillent ensemble, ce que l'autorisation signifie dans ce livre. Le grant décide de ce qu'un lecteur voit, et le chemin décide de la base de données que le lecteur peut atteindre tout court. Le chapitre traite aussi de ce qui découle de cette étape : chaque réplique, et toute autre copie des données, est un nouvel endroit où l'accès doit être conçu. La question est la suivante : quand les données sont lues à plus d'endroits qu'elles ne sont écrites, quelles lectures vont où, qui peut lire chaque copie, et comment empêcher les copies de se multiplier en portes dont personne n'est responsable ?

Le problème, et pourquoi il est difficile

Une base de données qui reçoit les écritures et sert aussi toutes les lectures fait deux métiers avec un seul jeu de ressources. Les écritures ont besoin du primary : il n'y en a qu'un, il est la source de vérité, et il doit rester disponible pour elles. Les lectures peuvent être répétées, elles peuvent être réparties, et beaucoup d'entre elles tolèrent des données vieilles de quelques secondes. Éloigner les lectures du primary protège les écritures, et protéger les écritures, c'est protéger le grand livre.

L'approche évidente consiste à ajouter une réplique et à y diriger les charges de travail riches en lectures. Cela fonctionne pendant une semaine environ. Ensuite, trois problèmes apparaissent, et la réplique elle-même n'en résout aucun.

Le premier est qu'une réplique n'est pas le primary. Dans la plupart des déploiements, la réplication est asynchrone : le primary fait le commit d'un changement, et la réplique l'applique peu après. Cet écart s'appelle le **retard de réplication**. Une lecture qui atterrit sur une réplique en retard renvoie une réponse vraie sur un monde un peu plus ancien. Pour un rapport mensuel, c'est invisible. Pour un client qui vient de payer une facture et regarde une page qui indique encore « à payer », c'est un ticket de support.

Le deuxième problème est que les privilèges voyagent avec les données. Une réplique physique est une copie complète, colonnes Restricted comprises, et tout ce qui peut lire la réplique peut lire ce qu'un lecteur du primary pourrait lire, sauf si les grants en disposent autrement. La section suivante explique pourquoi on ne peut pas rendre un rôle plus sûr sur une seule réplique.

Le troisième problème est que les copies se multiplient. La première réplique est pour l'application. La deuxième est pour le reporting, puis pour les investigations humaines, puis pour l'étape d'export. Un entrepôt de données suit, alimenté par une planification ou par change data capture. Quelqu'un restaure un snapshot dans une base de données de staging. Chaque copie était, le jour venu, une réponse raisonnable à un vrai problème de charge ou de commodité. Chacune contient aussi des données de production, porte ses propres credentials et son propre chemin réseau, et figure rarement sur la liste de qui que ce soit des voies d'entrée vers les

données. Une **porte latérale** est une copie de données ou un chemin d'export qui contourne les contrôles d'accès du primary. Ce chapitre porte sur les copies que l'on construit volontairement, et le Chapitre 13 revient sur celles qui apparaissent par accident.

Une politique qui dit « n'exécutez pas de rapports sur le primary » n'aide pas Jordan. Le client mémorise un seul hôte, le login fonctionne sur tous, et le primary est celui qui a toujours des données fraîches. Le contrôle doit se trouver dans la façon dont les lectures sont routées et dans ce à quoi chaque login peut se connecter.

Séparer le chemin de lecture

Je commencerais par traiter le chemin de lecture comme une surface d'accès à part entière, avec son propre inventaire, ses propres rôles et ses propres règles, et par poser une seule question à chaque lecture : cette lecture a-t-elle besoin du primary ? Une lecture en a besoin quand elle décide de quelque chose, comme un solde avant l'encaissement d'un paiement, ou quand elle doit voir une écriture faite un instant plus tôt dans la même session. Tout le reste peut aller vers une réplique. La façon dont un service classe et route ses propres lectures relève de la conception applicative, et ce livre la laisse de côté.

La règle d'accès est l'autre moitié, et elle tient en deux lignes. Les lectures de reporting et les lectures humaines ne se rabattent jamais sur le primary : si la réplique est indisponible, le job attend ou échoue et l'équipe plateforme est prévenue. Et les logins utilisés pour elles ne peuvent pas du tout se connecter au primary, si bien qu'une erreur ne peut pas les y envoyer. Une agrégation nocturne qui tourne avec deux heures de retard est un désagrément. Une agrégation qui bascule discrètement sur le primary parce que la réplique était en panne, c'est précisément ainsi que l'état initial est apparu.

Un rôle en lecture seule n'est pas un rôle inoffensif

On traite généralement les lectures comme de la charge. Elles sont aussi la principale exposition en matière de confidentialité : un jeu de résultats qui quitte le système ne change rien, ne laisse rien à annuler par rollback et peut être copié n'importe où. Un rôle qui ne peut pas écrire peut toujours

lire chaque colonne qui lui est accordée, et une seule requête lourde lancée avec lui peut faire tomber le primary. « Lecture seule » désigne aussi trois mécanismes différents, qui diffèrent selon que la session peut les défaire ou non.

Mécanisme	Ce qu'il empêche	La session peut-elle le défaire?
Un grant sans privilèges d'écriture	Les écritures de ce rôle	Pas à lui seul ; mais une session qui peut basculer vers un rôle doté de privilèges d'écriture n'est plus liée par lui
Une transaction en lecture seule par défaut	Les écritures dans cette session, par défaut	Sur de nombreux moteurs, oui : la session peut rétablir le paramètre ou démarrer une transaction en lecture-écriture
Une réplique physique	Toute écriture sur cet hôte	Non : l'hôte refuse les écritures, et la session n'a aucun paramètre pour changer cela

Pour les personnes, la réplique est le plus solide des trois, et c'est une des raisons pour lesquelles les lectures humaines courantes y vont. Ce qu'aucun d'eux ne limite, c'est la quantité qu'une session peut lire. Un row cap sur une session et des paliers pour les exports relèvent de l'Exécution, et le Chapitre 9 les couvre.

Les grants voyagent jusqu'à la réplique

La question sur les répliques que les relecteurs chargés de la sécurité posent en premier est « que peut-on y faire ? ». Sur les moteurs qui répliquent physiquement, les rôles et les grants vivent dans le catalogue du cluster, et ce catalogue est copié avec les données. Un rôle est défini une fois, sur le primary, et il apparaît sur chaque réplique. Une réplique est en lecture seule, si bien qu'on ne peut pas y créer, modifier ou accorder un rôle à elle seule. Il s'ensuit qu'on ne peut pas rendre un rôle plus sûr sur une seule réplique : réduire la surface d'un lecteur sur une réplique demande soit la couche de connexion, soit une copie séparée. Les privilèges que vous avez conçus au Chapitre 3 sont en vigueur sur chaque réplique, et la base de données qu'un login peut atteindre est une règle de la couche de connexion.

Cela a un bon côté. `human_readonly` retient les colonnes Restricted et masque les colonnes Confidential là où le Chapitre 12 le prévoit, et il le fait sur la réplique pour la même raison que sur le primary.

`app_reporting_job` lit sur une réplique sans aucun accès aux colonnes Restricted. Si les grants ont été construits avec le moindre privilège en tête, une réplique ne donne à un lecteur rien de plus que ce que le rôle permet.

Cela a aussi un côté risqué. Dans l'état initial, les répliques acceptent le login admin partagé, qui est propriétaire de toutes les tables. Un lecteur sur une réplique avec ce login voit chaque colonne, soit la même exposition que sur le primary moins la capacité d'écrire. On a tendance à considérer les répliques comme sûres précisément pour cette raison, et le résultat est que c'est sur les répliques que se trouve l'accès le plus large.

Diriger chaque login vers sa réplique

L'état cible donne à chaque réplique une finalité, et cette finalité est imposée là où les connexions sont acceptées. Deux termes sont employés dans un sens fixe à partir d'ici. Une **network policy** dit quels hôtes peuvent ouvrir une connexion TCP vers un hôte de base de données ; le Chapitre 7 en écrit une. Une **connection policy** dit quel login peut s'authentifier auprès de quelle base de données, et depuis où.

La connection policy a besoin d'un mécanisme, parce que les grants ne peuvent pas la porter et que la network policy ne voit pas les logins. Sur les moteurs qui en disposent, ce mécanisme est la règle d'accès basée sur l'hôte que chaque serveur lit dans sa propre configuration. Cette règle a pour clés la source de la connexion et un motif de login ou un rôle, et elle ne fait pas partie des données répliquées, si bien que le primary et chaque réplique peuvent porter des règles différentes. Le bloc ci-dessous est une telle politique, écrite comme les règles que porteraient les trois serveurs.

```
connection policy: one rule set per server, not replicated
    ledgerly_replica_1:
        allow from pooler hosts    login app_billing_api,
app_admin_console  // browsing, support lookups
        deny  everything else

    ledgerly_replica_2:
        allow from pooler hosts    login app_reporting_job,
app_vendor_export
        allow from gateway hosts   login pattern: kind readonly   //
human_readonly, Chapter 7
        deny  everything else

    ledgerly_primary:
        deny  from anywhere        login app_reporting_job,
app_vendor_export
        allow from gateway hosts   login pattern: kind jit, kind
breakglass
        deny  from gateway hosts   login pattern: kind readonly   //
routine reads use replica_2
```

Le coût est qu'il faut désormais garder trois jeux de règles en phase avec le schéma des rôles, et qu'un changement appliqué sur un serveur et pas sur un autre est une dérive. Les jeux de règles ont leur place dans le même pipeline que celui qui applique les définitions de rôles (Chapitre 3), et un contrôle qui compare les règles de chaque serveur à la politique a sa place dans la même revue. Là où un moteur n'a pas de telle règle, les mêmes décisions reviennent au pooler et à la gateway, ainsi qu'à la network policy qui les admet.

Associer `human_readonly` à `ledgerly_replica_2` signifie qu'une personne qui enquête sur un problème lit sur la même réplique que celle qu'utilise le reporting. C'est un échange délibéré : une seule réplique à protéger, surveiller et dimensionner, en contrepartie d'investigations humaines qui disputent la capacité au job nocturne. Si les deux commencent à se gêner, une troisième réplique pour les humains est une séparation raisonnable, et la connection policy change d'un bloc.

Les refus sur le primary comptent autant que les autorisations sur les répliques. `app_reporting_job` et `app_vendor_export` n'ont rien à faire sur le primary. Si leurs logins ne peuvent pas s'y connecter, le comportement « lit parfois le primary » de l'état initial n'est plus quelque chose qu'une er-

reur de configuration peut ramener. Un job dirigé vers le mauvais hôte n'arrive pas à se connecter, ce qui est une défaillance visible et peu coûteuse.

L'exception dans l'autre sens est une session JIT ou break-glass. Chaque login de session humaine est membre de `human_readonly` par défaut, si bien que la règle du primary ne peut pas être écrite contre ce rôle : soit elle couperait le break-glass, soit elle ne signifierait rien. Elle est écrite contre le type de session que l'access gateway a ouvert. Une session courante est envoyée vers `ledgerly_replica_2`. Une session qui doit écrire, ou lire en urgence ce qu'une réplique ne peut pas montrer assez vite, est ouverte sur le primary, et la demande d'accès est faite depuis cette session (les chapitres 7, 8 et 10 décrivent le modèle).

Cela laisse une fenêtre sur le primary : tant qu'une demande n'est pas approuvée, une telle session peut faire ce que fait `human_readonly`. Trois règles gardent cette fenêtre étroite, et le Chapitre 7 les énonce dans le cadre des types de session. Seules les personnes que la politique autorise à demander ce type peuvent l'ouvrir. Le motif est consigné à l'ouverture de la session, et la session a une courte fenêtre d'ouverture dans laquelle une demande doit suivre. Une alerte se déclenche si aucune ne suit. Une personne qui veut seulement regarder des données n'ouvre pas une telle session ; c'est à cela que sert la réplique, et c'est pourquoi les lectures courantes ne sont pas autorisées sur le primary.

La réplique a aussi un hôte

Une réplique physique est une copie des fichiers de données, si bien que ses hôtes, ses volumes, ses snapshots et ses credentials de réplication atteignent les mêmes données sans passer par un rôle de base de données. J'appliquerais aux répliques les mêmes règles d'accès aux hôtes, le même chiffrement du stockage et la même rétention qu'au primary, et je le consignerais à la création de la réplique, parce que personne n'y pensera plus tard. Le Chapitre 13 revient sur les snapshots et les sauvegardes.

Reporting, analytics et choix de la copie

Les répliques résolvent un problème de charge pour des lectures peu coûteuses et fréquentes. Le reporting est l'inverse : des lectures rares, volumineuses et lentes. La question est de savoir si le reporting a sa place sur une réplique ou ailleurs, et la réponse dépend de ce dont les rapports ont besoin.

Trois endroits où un rapport peut lire

Option	Fraîcheur	Charge sur le primary	Surface d'accès	Adapté à
Une réplique de reporting (`ledgerly_replica_2`)	De quelques secondes à quelques minutes	Réplication uniquement	Mêmes données et mêmes rôles que le primary	Rapports opérationnels, exports, investigations
Une copie séparée chargée selon une planification (`ledgerly_analytics`)	Aussi ancienne que le dernier chargement	Un chargement par planification	Ses propres rôles et ses propres données	Modèles, tableaux de bord, analyses en masse par l'équipe data
Une copie alimentée par change data capture	De quelques secondes à quelques minutes	Un lecteur du stream de changements	Un nouveau pipeline et un nouveau stockage	Analytics quasi temps réel, nombreux consommateurs

J'enverrais le reporting opérationnel vers une réplique et les analyses en masse vers une copie séparée, et je résisterais à l'idée de mettre le travail de l'équipe data sur la réplique pour une raison qui n'a rien à voir avec la charge. Une réplique porte la forme et la sensibilité du schéma de production. L'équipe data n'a pas besoin des colonnes Restricted et, dans l'état cible, ne voit pas non plus les colonnes Confidential brutes, parce que la copie d'analytics est masquée au chargement, à l'exception des indications d'affichage qui existent pour être montrées (le Chapitre 12 décrit comment et nomme l'exception). On ne peut pas masquer une réplique par consommateur sans changer ce qu'est la réplique. Une copie séparée peut être construite pour son public.

C'est aussi pourquoi la copie d'analytics n'est appelée réplique nulle part dans ce livre. Elle est chargée selon une planification depuis le primary, elle a des rôles différents, et son contenu peut différer de la source. L'appeler réplique invite à supposer qu'elle porte les mêmes contrôles, et dans l'état initial elle en porte moins.

Change data capture est l'option à laquelle je recourrais en dernier. Elle ajoute un pipeline qu'il faut maintenir en marche, un stockage qu'il faut sécuriser, un retard qui lui est propre et un contrat de schéma entre producteur et consommateur. Pour une entreprise de la taille de Ledgerly, je prendrais d'abord le chargement planifié et je consignerais ce qui me ferait changer d'avis : par exemple, des rapports qui ont besoin de données plus récentes que ce que permet la planification, ou une deuxième équipe avec le même besoin.

Le coût de chaque copie

Chaque option du tableau comporte le même poste caché : une nouvelle chose à protéger. Une copie est un endroit où vivent les données, un ensemble de logins qui peuvent les lire, un chemin réseau, une règle de rétention et un responsable. Une copie sans responsable est une porte latérale, et une copie avec un responsable mais sans expiration est une porte latérale qu'on met plus longtemps à remarquer.

Il est utile de tenir un court registre des copies, avec les mêmes champs pour chacune. Le bloc ci-dessous est une notation illustrative, et son intérêt est que les lignes `owner` et `ends` ne peuvent pas être vides.

```
table data_copy:
    name            text primary key    // "ledgerly_replica_2"
    kind            text not null       // replica, analytics, export,
dump, staging
    source          text not null       // what it was made from
    contains        text not null       // data classes present:
Restricted, Confidential, ...
    readers         text not null       // roles or logins that may
read it
    owner           text not null       // a team, never a person
    ends            timestamp null      // null only for replicas and
the analytics copy
    review_due      timestamp not null
```

Pour Ledgerly, le registre commence avec cinq entrées : les deux répliques, la copie d'analytics, les fichiers d'export qui partent chez le fournisseur comptable, et les vieux dumps du stockage partagé. Quatre sont volontaires et les dumps ne le sont pas, et c'est pour cela que le registre est utile : les dumps reçoivent un responsable et une date de fin proche. Le Chapitre 13 utilise le même registre pour les copies qui vivent en dehors des bases de données.

> **Remarque :** Un registre que personne n'ouvre n'est pas un contrôle, il est donc lié à l'étape qui crée les copies. L'étape de provisionnement refuse de créer une réplique, une restauration de snapshot, une base de données de staging ou un job d'export à moins qu'une ligne n'existe avec un responsable et, quand le type l'exige, une date de fin. La revue des accès (Chapitre 14) demande ensuite quelles copies existent et qui peut lire chacune.

Le refus a un coût : quelqu'un remplit une ligne avant qu'une équipe tenue par une deadline puisse obtenir sa copie, et les premières semaines produisent des plaintes. Il ne couvre aussi que les copies faites par l'étape de provisionnement, si bien qu'une copie faite à la main reste une porte latérale, et le Chapitre 13 s'en occupe.

La portée, pas seulement les grants

Un rôle correct sur le papier reste mauvais si le mauvais login peut se connecter. Pour chaque copie, le grant retient ce dont le public n'a pas besoin (chapitres 3 et 12), et la portée décide quels chemins réseau et quels principals peuvent ouvrir une connexion tout court. Sur un réseau plat, un login qui a fuité sur n'importe quel service atteint chaque copie, et la portée borne ce que vaut une fuite, comme le dit le Chapitre 4. Le tableau énonce la règle de portée pour chaque endroit ; la network policy qui l'impose côté hôte est celle du Chapitre 7, et la connection policy ci-dessus l'impose côté login.

Endroit	Accessible depuis	Ne doit pas être accessible depuis
`ledgerly_primary`	Les hôtes du pooler, la gateway pour les sessions JIT et break-glass, l'étape de pipeline qui applique les migrations	Les jobs de reporting et d'export, les sessions humaines courantes, la couche web directement
`ledgerly_replica_1`	Les hôtes du pooler, pour les services qui font de la navigation	La gateway pour les lectures courantes, le chargement d'analytics, la couche web directement
`ledgerly_replica_2`	Les hôtes du pooler pour le reporting et l'export, la gateway pour les lectures humaines courantes	Les services qui font tourner l'application, la couche web directement
`ledgerly_analytics`	Le job de chargement, `reporting-job` pour ses résultats, la gateway pour les sessions d'analytics	Les services qui font tourner l'application

Migrer le chemin de lecture de Ledgerly

Voyons comment Alex ferait passer Ledgerly de l'état initial à cette conception sans casser le job nocturne au premier essai. L'ordre compte plus que la vitesse, et je procéderais en cinq étapes.

1. **Inventorier les lectures.** Pour chaque service, listez les requêtes qu'il exécute et classez-les en lectures qui ont besoin du primary et lectures qui n'en ont pas besoin. La plupart des requêtes de listage de `billing-api` sont du second type. Ses écritures et ses redirections après écriture ne le sont pas. Les logs de requêtes sont la matière première des règles de routage.
2. **Donner une finalité à chaque réplique.** Écrivez la connection policy de `ledgerly_replica_1` et `ledgerly_replica_2` et appliquez-la d'abord aux nouveaux logins, en laissant pour l'instant le login partagé en place.
3. **Déplacer `reporting-job` et l'étape d'export.** Dirigez-les tous deux vers `ledgerly_replica_2` avec leurs propres logins, puis refusez ces logins sur le primary. Faites-les tourner en parallèle pendant quelques cycles et comparez les sorties avant de couper l'ancien chemin.
4. **Déplacer les lectures de navigation.** Routez les lectures de navigation de `billing-api` et `admin-console` vers `ledgerly_replica_1`

derrière un flag, et augmentez la part progressivement.

5. **Fermer le login partagé sur les répliques.** Une fois que plus rien de légitime ne l'utilise là-bas, retirez-lui la possibilité de se connecter à l'une ou l'autre réplique. C'est l'étape qui rend le reste permanent, et c'est celle que les équipes repoussent.

Le chemin de l'équipe data change aussi. La question de fin de mois de Jordan a désormais deux réponses. Si un chiffre de la veille convient, elle va vers `ledgerly_analytics` via `human_analytics_reader`. Si elle a besoin des données du jour, elle va vers `ledgerly_replica_2` via l'access gateway avec `human_readonly`, une session qui est journalisée, qui ne peut pas voir les colonnes Restricted et qui ne peut pas toucher le primary. Ce que Jordan perd, c'est la possibilité de diriger un client de bureau vers n'importe quel hôte et que cela fonctionne. Ce que Jordan gagne, c'est qu'une requête lourde est un problème pour la réplique et un problème mineur pour tous les autres.

Protéger une réplique de ses lecteurs

Une réplique qui sert des humains et un job nocturne verra passer des requêtes lourdes, et sur certains moteurs une longue requête peut retarder la réplication ou être annulée pour que la réplication puisse continuer, selon la configuration. Les contrôles sont ceux du Chapitre 5, appliqués aux rôles de la réplique : un timeout d'instruction plus court pour `human_readonly` que pour `app_reporting_job`, une limite de connexions par login que la gateway fixe sur chaque login de session humaine, et une étape d'export qui s'exécute en dehors de la fenêtre de l'agrégation nocturne. Ajoutez une alerte sur le retard de réplication, et envoyez l'appel d'astreinte au responsable de la réplique inscrit dans le registre. Une alerte que personne ne reçoit n'est pas un contrôle.

Compromis et coûts

Une réplique est un système à exploiter. Il faut la surveiller, la mettre à jour, la dimensionner et la restaurer, et pour une petite équipe une réplique et une copie nocturne peuvent suffire. L'argument d'accès tient même sans elle : vous décidez quels logins atteignent le primary et lesquels ne l'atteignent pas.

Partager `ledgerly_replica_2` a un prix. Les personnes et le job nocturne se disputent la capacité d'une seule réplique, en contrepartie d'un seul endroit à protéger. La séparation en une troisième réplique est bon marché en termes de politique, et coûteuse en exploitation.

La connection policy, c'est davantage de configuration. Les règles par serveur doivent correspondre au schéma des rôles et survivre à un failover, à une reconstruction et à une nouvelle réplique. La dérive entre serveurs est la défaillance, et le contrôle de comparaison est le prix à payer pour la garder visible.

Certains écrans seront en retard. Éloigner les lectures du primary signifie du retard de réplication, et quelqu'un le remarquera. Convenez à l'avance avec le support et le produit des écrans qui peuvent être en retard, et vérifiez d'abord le retard de réplication quand quelque chose ne colle pas dans une investigation sur une réplique (Chapitre 15).

Le registre ajoute de la friction. Exiger une ligne avant qu'une copie existe ralentit l'équipe qui voulait la copie, et ce ralentissement est l'effet recherché. Un registre que personne n'est obligé de remplir à la création est une liste de ce dont quelqu'un s'est souvenu.

Il y a des cas où les conseils de ce chapitre ne s'appliquent pas. Un petit système dont le primary a de la marge, et où les ingénieurs qui le lisent se connectent déjà par un chemin contrôlé, n'a peut-être pas besoin d'une réplique pour la charge, et séparer le chemin de lecture y rapporte moins. Un système dont chaque lecture a des exigences de fraîcheur très strictes devra peut-être exécuter la plupart des lectures sur le primary et consacrer ses efforts à limiter ce que ces lecteurs peuvent faire. Une réplique n'est pas non plus une sauvegarde ; le Chapitre 13 couvre les sauvegardes.

Erreurs courantes

- **Considérer une réplique comme moins sensible que le primary.** On ne peut pas y écrire, alors elle accumule les accès larges : le login partagé, un rôle de lecture permanent pour tout le monde, une règle de pare-feu plus large. Elle contient les mêmes données, donc elle reçoit les mêmes grants et une connection policy tout aussi étroite.

- **Laisser le reporting ou les lectures humaines se rabattre sur le primary.** Le repli est tentant parce qu'il fait tourner le rapport quand la réplique est en panne. Faites plutôt échouer la lecture, et refusez ces logins sur le primary pour qu'une erreur ne puisse pas les y envoyer.
- **Envoyer vers une réplique des lectures qui décident de quelque chose.** La lecture est peu coûteuse et la réplique est inactive, cela ressemble donc à un gain facile. Une décision prise sur des données périmées et suivie d'une écriture, c'est ainsi que les soldes et les statuts dérivent. Classez les lectures selon ce qui les suit, pas selon leur poids.
- **Construire un entrepôt de données en dirigeant un humain vers une réplique.** L'équipe data porte alors le schéma de la production, la sensibilité de la production et les limites de capacité de la production. Donnez à l'analytics sa propre copie et ses propres rôles (chapitres 12 et 13).
- **Ajouter des copies sans registre.** Chacune était une faveur faite à une équipe tenue par une deadline. Six mois plus tard, personne ne peut dire combien il en existe. Faites du responsable et de la date de fin une partie de la création d'une copie.
- **Oublier l'hôte et les snapshots.** Les rôles sont corrects, et le snapshot de volume de la réplique est lisible par quiconque a accès à l'infrastructure. Incluez l'hôte, le stockage et les snapshots dans les contrôles de chaque copie.
- **Supposer qu'un rôle peut être resserré réplique par réplique.** Sur les moteurs qui répliquent physiquement, les rôles valent pour tout le cluster. Réduisez la surface d'un lecteur au niveau de la couche de connexion ou avec une copie séparée.

Points clés

- Les écritures ont besoin du primary et la plupart des lectures non, donc la première question pour le chemin de lecture est de savoir si une lecture décide de quelque chose ou suit une écriture ; les lectures de reporting et les lectures humaines ne se rabattent jamais sur le primary.
- « Lecture seule » recouvre trois mécanismes, et seule la réplique ne peut pas être défaite par une session, donc un rôle en lecture seule limite les écritures et non ce qu'une session peut lire.

- Une réplique contient les mêmes données que le primary et est régie par les mêmes rôles à l'échelle du cluster, donc les grants du Chapitre 3 et la connection policy décident ensemble de ce qu'un lecteur peut voir.
- La connection policy est une règle par serveur dont les clés sont la source et le login, tenue en phase avec le schéma des rôles ; chaque réplique a une seule finalité, et le login admin partagé n'est accepté sur aucune d'elles.
- Une session JIT ou break-glass sur le primary exige une personne autorisée à l'ouvrir, un motif consigné, une fenêtre courte et une alerte quand aucune demande ne suit.
- L'analytics a sa place sur une copie séparée construite pour son public, et l'appeler réplique masque le fait qu'elle porte des contrôles différents.
- Chaque copie de données a besoin d'un responsable, d'une liste de lecteurs et, sauf pour les répliques et la copie d'analytics, d'une date de fin, et l'étape qui crée les copies refuse de s'exécuter sans eux.

Le chemin de lecture a désormais ses propres rôles, ses propres destinations et une courte liste de copies, mais nous n'avons pas encore dit comment une personne atteint l'une d'elles. Le chapitre suivant aborde les chemins d'entrée pour les humains, et compare un bastion, un proxy de base de données et une access gateway.

PARTIE III

L'accès humain

Chapitre 7 — Les chemins d'entrée: bastions, proxies et access gateways

Imaginez qu'Alex, de l'équipe plateforme, se voie poser une question simple pendant une réunion de planification : « Si nous coupions demain un chemin d'entrée vers la base de données de production, lequel ferait mal ? » Personne ne répond vite. Alex part chercher la réponse, et commence par le réseau. Le VPN donne à chaque ingénieur une route vers le primary. Il y a aussi un hôte bastion que quelqu'un a mis en place pour un prestataire il y a deux ans, et il a toujours un port en écoute. Les clients de bureau d'une douzaine d'ordinateurs portables gardent des connexions enregistrées ; l'une d'elles a été configurée pour passer par un tunnel à travers le bastion, tandis qu'une autre passe directement par le VPN.

Alex demande ensuite à Robin, d'astreinte cette semaine, comment Robin atteindrait la base de données la nuit. Robin cite deux chemins sans hésiter, puis un troisième après réflexion. Chaque chemin fonctionne, aucun n'a été conçu, et chacun aboutit au même login admin partagé. L'équipe réseau peut dire qui est sur le VPN. La base de données peut dire que le login admin s'est connecté. Personne ne peut dire quelle personne se trouvait derrière une connexion donnée, et personne ne peut couper l'accès d'une personne sans couper celui de tout le monde.

La question de ce chapitre est donc la suivante : qu'est-ce qui doit se tenir entre une personne et la base de données, pour qu'atteindre le réseau et atteindre les données soient deux décisions distinctes, chacune avec un nom attaché ? Ce chapitre ajoute l'access gateway, la première étape du versant humain de l'état cible de Ledgerly. Parmi les cinq parties du livre, elle porte le Chemin, et l'identité de quiconque y entre : le réseau décide qui peut frapper à la porte, et la gateway met un nom sur chaque session. Les rôles qu'une session de la gateway reçoit, et la façon dont une personne demande davantage que la lecture, relèvent du Chapitre 8.

Le problème, et pourquoi il est difficile

Un chemin réseau et un chemin vers la base de données semblent être la même chose, parce que le premier est ce dont vous avez besoin avant de pouvoir tenter le second. Une personne qui est sur le VPN peut ouvrir une connexion vers le port de la base de données. Ce que cette personne a ensuite le droit de faire est décidé par un nom d'utilisateur et un mot de passe, et dans l'état initial de Ledgerly le mot de passe est partagé. Le réseau a dit « vous pouvez frapper », et la base de données ne pouvait pas savoir qui avait frappé.

C'est le cœur du problème. Les contrôles réseau répondent à la question « quelles machines peuvent parler à quelles machines », et ils le font bien. Ils ne répondent pas à « quelle personne est-ce, qu'a-t-elle le droit de faire en ce moment, et qu'a-t-elle fait ». Quand on demande à une règle réseau de tenir lieu de réponse à ces questions, elle échoue sans bruit : la règle reste correcte tandis que l'accès qu'elle accorde devient périmé. Un ingénieur qui a changé d'équipe est toujours sur le VPN. Un prestataire dont la mission a pris fin est toujours dans un groupe que personne ne revoit.

Trois autres choses rendent le chemin humain plus difficile que le chemin des services.

Premièrement, les personnes sont les clients les plus variés. Un service se connecte depuis un endroit connu avec un driver connu. Une personne peut utiliser un client en ligne de commande, une interface graphique de bureau, un notebook, un outil dans le navigateur, ou un script écrit mardi dernier. Chacun a sa propre façon de mémoriser les credentials, son propre réglage par défaut pour l'auto-commit et sa propre habitude de télécharger les résultats sur le disque local. Le Chapitre 1 a listé les clients qui mémorisent le mot de passe et font de l'auto-commit ; ce n'est pas une configuration inhabituelle, c'est le comportement par défaut de nombreux outils conçus pour le confort.

Deuxièmement, les personnes se connectent depuis les endroits les moins contrôlés. Un ordinateur portable voyage, se met en veille, est sauvegardé et parfois partagé à la maison. Une connexion enregistrée qui contient un mot de passe se déplace avec l'ordinateur portable, tout comme chaque jeu de résultats exporté dans un fichier. La base de données ne voit rien de tout cela.

Troisièmement, le chemin est utilisé sous pression. L'ingénieur qui a besoin de la base de données la nuit n'est pas d'humeur à apprendre un nouvel outil, et un chemin lent ou fragile sera contourné par quiconque en a les moyens. Une conception correcte sur le papier qui perd face à l'ancien raccourci à 3 h du matin n'a rien changé, et c'est pourquoi le raccourci doit être supprimé, et pas seulement découragé.

Une politique qui dit « ne partagez pas le login admin » ne peut pas être vérifiée, parce que le log de la base de données montre le login admin et rien d'autre. Une conception dans laquelle chaque session a son propre login peut être vérifiée, tout comme une règle réseau qui ne laisse qu'une seule chose atteindre la base de données. Le reste de ce chapitre compare les façons de construire une telle conception.

Quatre chemins d'entrée, comparés

Avant la comparaison, deux termes, parce que le secteur emploie ces mots de façon approximative et que le livre a besoin de les fixer.

Un **bastion** est un hôte durci qui constitue le chemin réseau unique vers un système protégé. Il transfère des connexions. Il ne comprend pas le protocole de la base de données, si bien qu'il ne peut pas distinguer un `SELECT` d'un `DROP`, et il ne sait pas qui se trouve derrière la connexion, à moins que la personne ne se soit connectée au bastion lui-même.

Une **access gateway** est un intermédiaire qui connaît l'identité : elle authentifie une personne, émet un credential de courte durée et enregistre la session. Un **proxy de base de données** est autre chose : un intermédiaire qui connaît le protocole et qui mutualise, route ou filtre les connexions à la base de données. Une gateway peut être construite au-dessus d'un proxy de base de données, mais les deux répondent à des questions différentes. Un proxy connaît le protocole ; une gateway connaît la personne. Ce chapitre emploie chaque terme uniquement dans ce sens.

Voici les chemins qu'une équipe envisage habituellement, avec ce que chacun sait de la personne et ce qu'il laisse derrière lui. Les quatre premiers placent une personne dans une session ; les deux derniers l'en tiennent à l'écart.

Chemin	Ce qu'il sait de la personne	Ce que montre le log de la base de données	Coût principal
Endpoint public	Rien au-delà du login de la base de données	Un login, depuis n'importe quelle adresse	Expose le port de la base de données à tout Internet
VPN vers la base de données	Que la personne est sur le réseau	Le login partagé ou personnel	L'appartenance au réseau est traitée comme une identité
Bastion	Qui s'est connecté à l'hôte bastion, si les comptes sont personnels	Le login que la personne a saisi, depuis l'adresse du bastion	Un hôte de plus à patcher, et il ne voit aucune instruction
Access gateway	L'identité issue du single sign-on de l'entreprise	Un login de session qui correspond à une personne et à une session	Un nouveau système qui doit être disponible quand quelqu'un en a besoin
Chemin produit, sans SQL	Le rôle de la personne dans un outil interne	Le login propre du service, avec la personne dans la piste d'audit de l'outil	Chaque tâche exige qu'une fonctionnalité soit construite ; les questions libres n'ont aucun chemin
Exécuteur à soumission	La personne qui a soumis, et le relecteur qui a approuvé	Le login de l'exécuteur, qui exécute un ensemble d'instructions relu	Chaque tâche attend une revue ; rien n'est interactif

Les deux dernières lignes ne sont pas du tout des façons d'atteindre une base de données. La personne ne détient jamais de session, et l'instruction s'exécute depuis un composant qui l'a d'abord vérifiée. Elles ont leur place dans la comparaison parce que, pour de nombreuses tâches, elles sont la meilleure réponse : une personne qui n'ouvre jamais de session ne peut pas en laisser une ouverte. Le Chapitre 1 a présenté l'éventail de ces patterns, le Chapitre 9 couvre la façon dont la gateway vérifie une instruction, et le Chapitre 10 couvre les patterns de correction de données ; ce chapitre couvre les sessions qui restent une fois que les tâches adaptées à ces patterns sont parties ailleurs.

L'endpoint public

Je n'exposerais pas la base de données à l'Internet public pour un usage humain, et l'argument est court. Un port accessible attire un trafic automatisé qui n'a rien à voir avec les personnes de Ledgerly, et chaque credential

de cette base de données devient alors quelque chose qui vaut la peine d'être deviné. La seule défense qui reste est la solidité du login, et la prémisse du livre est que les logins fuient.

Il existe de rares cas où une base de données managée n'est accessible que publiquement, et où le fournisseur propose une allowlist d'adresses sources. Si vous êtes dans cette situation, traitez l'allowlist comme le minimum et passez à un chemin privé dès que le fournisseur le permet. Le test est simple : quelqu'un qui n'a jamais travaillé pour vous pourrait-il ouvrir une connexion vers le port ? Si oui, chaque autre contrôle de ce livre part d'une position moins favorable.

Le VPN

Un VPN est une couche réseau raisonnable. Il restreint l'accès à la base de données aux personnes à qui l'on a donné une route, et la plupart des entreprises en ont déjà un. Sa limite est ce qu'il prouve. Il prouve qu'une personne, ou l'ordinateur portable d'une personne, a passé les vérifications du VPN. Il ne dit pas quel login de base de données elle a utilisé ensuite, et il n'enregistre pas les instructions.

Dans l'état initial de Ledgerly, le VPN est tout le contrôle. Chacun de ses utilisateurs peut atteindre le primary, et le login admin est le même pour tous. Je garderais le VPN, mais je lui confierais un rôle plus restreint : il amène les personnes à la gateway et aux autres outils internes, et il ne les amène plus au port de la base de données.

Le bastion

Un bastion ajoute une propriété utile : un endroit unique où la porte réseau peut être fermée. Le pare-feu de la base de données n'autorise que l'adresse du bastion, si bien que désactiver un compte du bastion coupe le chemin réseau d'une personne. Si le bastion a des comptes personnels et enregistre qui s'y est connecté, vous gagnez un relevé approximatif de qui était connecté et quand.

Ce qu'un bastion ne vous donne pas, c'est quoi que ce soit en dessous de la couche réseau. Une fois qu'une personne a un tunnel ouvert à travers lui, la connexion à la base de données est ce que le client de la personne en fait. Le login est toujours saisi par la personne ou enregistré dans son client. Les instructions sont invisibles pour le bastion. Un bastion met donc

de l'ordre dans le tableau réseau sans réparer le tableau des identités, et c'est pourquoi une équipe peut en avoir un et rester incapable de répondre à « qui a exécuté cette instruction ? ».

Un bastion a aussi un problème de maintenance discret. C'est un hôte avec un shell, il a donc besoin de patchs, d'une gestion des clés et d'un responsable qui se souvient qu'il existe. L'hôte du prestataire de l'ouverture est la forme habituelle de la défaillance : il était correct le jour où il a été construit, et personne ne lui a été affecté ensuite.

L'access gateway

L'access gateway est l'endroit où j'enverrais les personnes, pour les raisons qui suivent.

Elle authentifie la personne auprès du single sign-on de l'entreprise, si bien que l'identité est celle que l'entreprise contrôle déjà. Quitter l'entreprise, changer d'équipe et échouer à une vérification multifacteur prennent tous effet au niveau de la gateway sans que personne ne modifie la base de données. Elle émet un credential qui appartient à une seule session et meurt avec elle, si bien qu'il n'y a aucun mot de passe à enregistrer. Et elle enregistre la session, si bien que « qui a exécuté cette instruction ? » a une réponse à deux endroits : l'enregistrement de session de la gateway et le log de la base de données, qui nomme un login correspondant à exactement une personne.

Les gateways se présentent sous deux grandes formes. Certaines se placent dans le chemin des données : le client de la personne se connecte à la gateway, et la gateway ouvre la connexion à la base de données pour le compte de la personne. D'autres se contentent de fournir le credential et s'écartent, si bien que le client se connecte directement à la base de données avec le login de courte durée. La première forme peut enregistrer les instructions et appliquer des règles à la connexion. La seconde est plus simple et place moins de choses dans le chemin, mais le réseau doit alors permettre aux personnes d'atteindre le port de la base de données, ce qui ramène une partie du problème par lequel ce chapitre a commencé.

Pour Ledgerly, je choisirais la forme dans le chemin des données, parce qu'elle permet à la règle réseau d'être absolue : seule l'adresse de la gateway peut atteindre les hôtes de base de données. Le prix est que la gate-

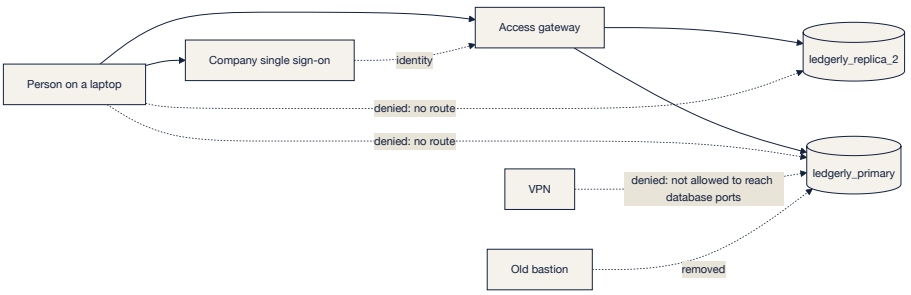
way traite chaque instruction humaine, si bien que sa capacité et sa disponibilité comptent. J'y reviens plus bas.

Séparer la portée réseau de l'accès à la base de données

Le changement le plus utile de ce chapitre n'est pas un produit. Il consiste à séparer deux questions auxquelles l'état initial répond ensemble.

La première question est la **portée réseau** : quelles machines peuvent ouvrir une connexion vers quel hôte et quel port de base de données. La seconde est l'**accès à la base de données** : quel login peut faire quoi, une fois connecté. Dans l'état initial, le VPN répond à la première et un mot de passe partagé répond à la seconde. Après le changement, une règle réseau répond à la première pour tout le monde à la fois, et la gateway répond à la seconde pour chaque personne séparément.

Le diagramme montre l'organisation cible pour les humains. Regardez les flèches refusées : ce sont les chemins qui existent aujourd'hui et qui n'existeront plus ensuite.



Deux points d'application se trouvent derrière ce diagramme. La règle réseau sur les hôtes de base de données n'autorise les connexions que depuis la gateway, depuis les réseaux propres des services, depuis le pipeline et depuis le credential broker. Et la gateway ne transmet une session qu'à la base de données que le type de la session autorise : une session de lecture courante va vers `ledgerly_replica_2`, et une session ouverte pour faire une demande JIT ou break-glass va vers `ledgerly_primary`, comme l'a établi le Chapitre 6.

J'écrirais la règle réseau sous forme d'une courte liste que tout le monde peut lire, parce que c'est la liste que vérifie une revue des accès. Le Chapitre 6 a défini la différence : cette liste est la network policy, qui décide quels hôtes peuvent se connecter tout court, tandis que la connection policy décide quel login peut s'authentifier auprès de quelle base de données. Si elle contient une entrée que personne ne sait expliquer, cette entrée est un chemin.

```
network policy (illustrative):
    ledgerly_primary:
        allows  access gateway
        allows  service networks of billing-api, invoice-worker,
                payments-service, admin-console
        allows  deployment pipeline
        allows  credential broker, for access administration
        denies  everything else, including the VPN

    ledgerly_replica_1:
        allows  service networks of the read-only services
        allows  replication traffic from the primary
        denies  everything else, including the VPN

    ledgerly_replica_2:
        allows  access gateway
        allows  reporting-job network
        allows  replication traffic from the primary
        denies  everything else, including the VPN

    ledgerly_analytics:
        allows  analytics-job network
        allows  access gateway, masked views only
        denies  everything else, including the VPN
```

La même liste continue avec ce qui sort des hôtes de base de données et ce qui peut atteindre la gateway.

```
network policy (illustrative, continued):
    egress from the database hosts:
        allows  replication, the audit log, backups
        denies  everything else

    the gateway's own exposure:
        reachable from  the VPN and the office network, nothing else
        reaches         the database hosts, the audit log, single
sign-on
```

Remarquez le peu de travail qui reste au VPN dans ce modèle. Il ne garde plus la base de données. Il amène les personnes à la gateway, et la gateway décide du reste. Le VPN compte moins ; l'accès de chaque personne est désormais lié à une session nommée.

Deux entrées de la liste sont faciles à oublier. L'egress vient en premier. Un hôte de base de données qui peut ouvrir des connexions sortantes vers n'importe où peut servir à faire sortir des données, et une courte allowlist (réplication, log d'audit, sauvegardes) transforme cela en un événement visible. L'exposition propre de la gateway vient en second : elle est désormais la seule porte accessible, elle figure donc elle-même sur une courte liste, et elle ne doit être accessible depuis aucun endroit que le VPN n'atteint pas déjà.

La liste suppose aussi que le réseau sait distinguer les appelants. Sur un réseau de cluster partagé, où plusieurs services viennent de la même plage d'adresses, il ne le peut pas : une règle qui autorise « les réseaux des services » les autorise tous. Dans ce cas, la règle réseau tient à l'écart tout ce qui n'est ni un service ni la gateway, et ce sont le pooler et le login qui distinguent un service d'un autre (chapitres 5 et 6). Je l'écrirais comme la règle du cluster, pour que personne ne lise la liste réseau comme plus précise qu'elle ne l'est. Quel login peut s'authentifier auprès de quelle base de données relève de la connection policy du Chapitre 6, et ce chapitre se contente d'y renvoyer.

L'access gateway chez Ledgerly

Voici ce que fait la gateway, dans l'ordre où elle le fait, et ce qui se passe quand une étape échoue. C'est la partie que je testerais soigneusement, parce qu'une gateway qui échoue en mode ouvert devient le nouveau mot de passe partagé, avec une meilleure image de marque.

Ce qu'est une session

Une session commence quand une personne s'authentifie. La gateway vérifie l'identité, décide quel type de session la personne a le droit de demander (une session de lecture courante pour la plupart des gens ; les types JIT et break-glass uniquement par les demandes du Chapitre 8), puis crée un **login de session**, le compte qu'utilise la session de la personne. Il est

créé sur `ledgerly_primary` et se réplique (Chapitre 6). La session se connecte ensuite à la base de données à laquelle appartient son type. Ce login porte l'identité de la personne dans son nom, si bien qu'il apparaît dans le log de la base de données, et il est membre de `human_readonly` par défaut. Son credential est généré pour cette session, remis à la gateway et jamais montré à la personne, si bien qu'il n'y a rien à enregistrer.

Un nom de login a une forme fixe, pour qu'une ligne de log puisse être lue sans consultation : un préfixe qui indique qu'il s'agit d'un login de session, l'identifiant court de la personne et le numéro propre de la session, comme dans `sess-robin-0412`. Le nom identifie ; c'est l'enregistrement de la gateway qui le relie à la personne et à la demande, et rien ne devrait analyser le nom pour décider de ce que la session peut faire.

Pour créer et supprimer des logins, et pour accorder et révoquer des appartenances à des rôles, la gateway a besoin de privilèges dans la base de données. Elle les obtient par son propre login pour le rôle `ledgerly_access_admin`. L'étape de pipeline des définitions de rôles et le credential broker utilisent aussi ce rôle, chacun par son propre login (chapitres 3 et 4). Cela fait du login de la gateway un credential sensible : quiconque le détient peut accorder n'importe quoi à l'intérieur des rôles qu'il administre. Je le garderais au même endroit et sous les mêmes règles que les credentials du propriétaire, je ne lui donnerais aucun privilège sur les données, et j'auditerais chacune de ses utilisations (Chapitre 14). Le copier dans des scripts ou dans un chat par commodité recréerait le mot de passe partagé un niveau plus haut.

Qui peut ouvrir quel type de session

`mayRequestKind` relève de la politique, et je le garderais sous forme de données qu'un relecteur peut lire, pas sous forme d'une condition enfouie dans le code de la gateway. Une seule table y répond : le groupe du single sign-on, les types que ce groupe peut ouvrir, la durée maximale de chacun, la base de données sur laquelle il aboutit, ses valeurs par défaut (lecture seule, timeout d'instruction, row cap) et la possibilité d'exporter ou non.

```
session policy (illustrative):
    group backend-engineers:
        routine       max 8 hours, ledgerly_replica_2, read-only, row
cap, no export
        primary-bound max 2 hours, ledgerly_primary, only to file a
request
    group analysts:
        routine       max 8 hours, ledgerly_analytics, masked views,
export logged
    group contractors:
        routine       max 4 hours, ledgerly_replica_2, masked views,
no export
```

Les types destinés au primary sont les plus sensibles, puisqu'une session sur le primary n'est qu'à une demande de l'accès en écriture. Quatre règles les gardent rares : seuls des groupes nommés peuvent en ouvrir une ; elle doit être ouverte avec un motif consigné ; la gateway n'attend que peu de temps, disons quinze minutes, qu'une demande soit déposée depuis la session ; et une session qui se termine sans demande déclenche une alerte, parce que « a ouvert le primary et n'a rien fait » est un pattern qui mérite qu'on le remarque. Le plafond de deux heures ne vaut que jusqu'à un grant : quand `grantJitAccess` ou `declareBreakGlass` du Chapitre 8 approuve une demande, la gateway remplace le `ends_at` de cette session par le `expires_at` de la demande, et `endSession` ci-dessous ferme la session à ce moment-là. Le Chapitre 8 part de l'autre côté, avec la demande.

L'ordre des opérations

La gateway, le log d'audit, la base de données et le single sign-on sont des systèmes distincts. Aucune transaction ne les englobe, si bien que l'ordre des étapes et le traitement d'une session à moitié ouverte doivent être conçus.

```
function openSession(person, wantedKind):
    identity = verifySingleSignOn(person)
    if not mayRequestKind(identity, wantedKind):
        raise SessionRefused
    login = newLoginName(identity)
    endsAt = now() + maxSessionLength(wantedKind)

    // the record comes first: a login with no record is what the
sweep hunts
    recordSession({ login: login, identity: identity, kind:
wantedKind,
                    ends_at: endsAt, status: "opening" })
    try:
        auditLogAppend({ actor: identity, action: "session-open",
                         subject: login, ends_at: endsAt })
    on AuditLogUnavailable:
        markSessionRefused(login)
        raise SessionRefused

    try:
        // always on the primary: logins are cluster-wide and
replicate
        createLogin(login, on: "ledgerly_primary", password:
freshSecret())
        grant("human_readonly", to: login)
        // replication lag: wait until the target database knows the
login
        waitUntilLoginVisible(login, on: databaseFor(wantedKind))
    on DatabaseError:
        // a login with no role is harmless, but it must not linger
        dropLoginIfExists(login)
        markSessionRefused(login)
        raise SessionRefused
    markSessionOpen(login)
    return connectionFor(login, databaseFor(wantedKind))
```

Lisez le bloc pour son traitement des défaillances. L'enregistrement est écrit avant que le login n'existe, si bien qu'un crash au milieu laisse soit un enregistrement qui indique « opening » sans login, soit un login que l'enregistrement explique. L'entrée d'audit vient avant le login : si le stockage du log est injoignable, aucune session ne s'ouvre. Ce choix a un coût, que la section sur les compromis nomme. Pour une session de lecture courante, la gateway attend que `ledgerly_replica_2` ait reçu le login, car sinon le retard de réplication ferait échouer de temps à autre la première connexion. Si la création du login ou l'attribution du rôle échoue, la gateway tente de nettoyer et refuse la session. Si même le nettoyage échoue, un login sans privilèges existe, et le sweep ci-dessous le supprime.

Terminer une session

Une session se termine de l'une de trois façons : la personne se déconnecte, la limite d'inactivité est dépassée, ou `ends_at` est dépassé. Les deux dernières comptent parce que les gens laissent des onglets ouverts. Une connexion ouverte survit à un credential expiré (Chapitre 4), si bien que la gateway ferme elle-même les connexions, puis supprime le login. La même fonction sert pour une personne qui quitte l'entreprise : un événement de déprovisionnement venu du single sign-on appelle `endSession`, et la durée maximale de session est la borne pour le cas où cet événement n'arrive jamais.

```
function endSession(login, reason):
    // the single owner of a session's end; Chapter 8's expiry calls
it too
    try:
        closeConnections(login)
    on CloseFailed:
        // a drop that fails on a live connection is retried by the
sweep
        notify("platform", "connections not closed: " + login)
    // requestsOf returns only requests not yet marked revoked
    for each id in requestsOf(login):
        try:
            revokeRequestAccess(id, from: login)
        on LoginNotFound:
            // nothing is left to revoke
        on RevokeFailed:
            notify("platform", "session grants not revoked: " + login)
            return
        markRevoked(id)
        auditOrQueue({ action: "revoke", actor: "gateway",
                       subject: login, request: id })
    try:
        revokeIfExists("human_readonly", from: login)
        dropLoginIfExists(login)
    on DatabaseError:
        notify("platform", "session login not removed: " + login)
        return
    markSessionEnded(login, reason)
    auditOrQueue({ action: "session-end", subject: login, reason:
reason })
```

Ici, `auditOrQueue` ajoute l'enregistrement au log d'audit et, si le log est injoignable, le met en file d'attente pour un nouvel essai et prévient l'équipe plateforme. La révocation et la suppression ont déjà eu lieu à ce stade, si bien qu'une panne du log retarde l'entrée et jamais le retrait de l'accès. Un login déjà disparu est traité comme un succès : une session que le chemin

d'expiration du Chapitre 8 a terminée plus tôt est fermée proprement ici, son enregistrement est marqué comme terminé et aucune alerte ne se déclenche.

Le sweep appelle `endSession` pour chaque session qui devrait déjà être terminée.

```
function sweepSessions():
    // endSession reports a stuck login and returns, so one failure
does not stop the loop
    for each s in sessionsPastEnd(before: now()):
        endSession(s.login, "expired")
    for each s in sessionsStuckOpening(before: now() - openingGrace):
        endSession(s.login, "abandoned")
    for each login in loginsWithoutRecord():
        endSession(login, "orphaned")
```

L'ordre est l'essentiel. Les grants d'une demande sont révoqués à partir de son enregistrement stocké et marqués comme révoqués seulement une fois la révocation réussie, et c'est seulement ensuite que le login est supprimé, parce que certains moteurs refusent de supprimer un login qui détient encore des privilèges. Une révocation ou une suppression qui échoue prévient l'équipe plateforme et laisse l'enregistrement au sweep suivant ; une écriture d'audit qui échoue est mise en file d'attente et n'arrête pas la boucle. Le sweep accorde un délai de grâce à une session encore à l'état « opening », pour ne pas supprimer un login qui n'a que quelques secondes. La vérification de dérive du Chapitre 3, qui compare les logins existants aux enregistrements de la gateway et les grants aux triplets stockés, tourne selon sa propre planification, si bien qu'un job arrêté ne masque pas l'autre.

Quelle base de données, quel rôle

Chaque session aboutit là où son type a sa place. Une session courante est un login `human_readonly` qui se connecte à `ledgerly_replica_2`. Une session ouverte pour faire une demande JIT ou break-glass est ouverte sur `ledgerly_primary`, et le Chapitre 8 montre comment la demande est faite depuis cette session et comment les grants prennent fin avec elle. La gateway applique la règle de destination que le Chapitre 6 a écrite sous forme de connection policy : c'est la gateway qui connaît le type, et c'est l'hôte de base de données qui refuse un login destiné à un autre.

Une personne peut avoir deux sessions ouvertes en même temps, l'une courante et l'autre avec une demande. Je garderais cela possible et je garderais les sessions visiblement séparées, pour que personne n'utilise la session privilégiée pour une consultation anodine. Le coût humain est une deuxième connexion dans le client de la personne. Le bénéfice est que la session privilégiée est facile à repérer, courte et facile à terminer.

Ce que la gateway enregistre

Comme la gateway de Ledgerly se trouve dans le chemin des données, elle peut enregistrer chaque instruction qu'une session envoie, ainsi que le début, la fin et le nombre de lignes affectées. Je stockerais cet enregistrement là où les personnes qu'il décrit ne peuvent pas le modifier ; le Chapitre 14 dit où. Un enregistrement ressemble à ceci :

```
statement record (illustrative):
    login:         sess-robin-0412
    person:        robin
    kind:          jit
    request:       req-0412
    fingerprint:   UPDATE invoices SET status = ? WHERE id IN (?)   //
literals redacted
    param_hash:    9c1e...
    rows:          12
    started_at:    03:21:07
    ended_at:      03:21:07
```

Les régimes applicables aux données de paiement et au reporting financier ont tendance à demander qui a touché aux données de production et par quel chemin ; l'enregistrement de session est cette réponse, et ce sont les collègues des services juridique et conformité du lecteur qui en ont l'interprétation. Deux limites méritent d'être connues. Un log d'instructions vous dit ce qui a été demandé, pas ce qui a été renvoyé, et je ne stockerais pas les jeux de résultats complets, parce qu'ils feraient du log une copie des données. Le fingerprint porte la forme de l'instruction et un hash de ses paramètres au lieu de leurs valeurs, pour la même raison. Et un log que personne ne lit enregistre sans rien protéger, alors décidez qui le regarde, et quand : un échantillon hebdomadaire des sessions qui ont détenu une demande JIT est un début, et la revue des accès est l'autre occasion.

La capacité et le budget de connexions

Chaque session humaine est aussi une connexion à la base de données, et le budget du Chapitre 5 a donné aux humains une part fixe, disons vingt connexions, à titre d'illustration. Les sessions courantes des ingénieurs aboutissent toutes sur `ledgerly_replica_2`, si bien que cette réplique est l'hôte de la part humaine, et sa limite de connexions est l'endroit où le chiffre est appliqué en dernier ressort. La gateway devrait appliquer elle-même cette part : une limite par personne, une limite par type de session, et un refus clair quand le budget est épuisé. Un refus au niveau de la gateway est un message adressé à une personne. Un refus au niveau de la base de données, quand le budget est épuisé, peut évincer un service. Placer la limite dans la gateway évite que le budget humain devienne le problème du service.

L'ordinateur portable et la copie locale

La gateway contrôle le chemin vers la base de données. Elle ne contrôle pas ce qui arrive aux données une fois qu'elles ont atteint un ordinateur portable, et c'est là que se loge une part discrète du risque.

Deux choses restent sur l'ordinateur portable. L'une est la configuration de connexion : l'hôte enregistré, le login enregistré et, dans l'état initial, le mot de passe enregistré. L'autre est tout ce que la personne fait d'un résultat : un fichier exporté, un tableau collé, une copie locale de quelques milliers de lignes pour « faire des essais ». Chacune est une copie de données au sens du Chapitre 6, un endroit où vivent des données de production sans responsable ni expiration.

Je traiterais ces choses en trois couches, de la moins coûteuse à la plus coûteuse.

Supprimer ce que le client peut mémoriser. Comme les credentials de session sont générés pour une seule session et jamais montrés, un client de bureau n'a aucun mot de passe à enregistrer. La connexion enregistrée contient désormais une adresse de gateway et le single sign-on de la personne, et les anciennes entrées enregistrées pour le primary peuvent être supprimées pendant la migration, avec une courte liste de ce qu'il faut supprimer et où. Cette couche ne coûte presque rien et supprime la fuite la plus courante.

Choisir des valeurs par défaut adaptées à la tâche. La gateway peut ouvrir des sessions avec des valeurs par défaut en lecture seule et un timeout d'instruction tiré du Chapitre 5. Elle peut aussi plafonner les lignes qu'une instruction renvoie : elle compte les lignes à mesure qu'elle les transmet, ferme le curseur au plafond, en informe la personne et écrit l'événement dans l'enregistrement d'instruction ; une session qui atteint le plafond de façon répétée est la première chose que regarde l'échantillon hebdomadaire.

Des valeurs par défaut que la personne peut lever ne restent pas des valeurs par défaut. Un client peut envoyer un `SET` qui augmente le timeout ou désactive la lecture seule, si bien que la gateway tient une liste de guarded settings et refuse toute instruction qui les modifie. Le coût est que certains clients envoient de telles instructions à la connexion et vont casser ; c'est un coût à nommer dans la migration, et le Chapitre 9 couvre la façon dont la gateway décide quelles instructions refuser. Les clients qui font de l'auto-commit sont moins dangereux quand la session n'a aucun privilège d'écriture à valider. Une console dans le navigateur, exécutée par la gateway elle-même, va un cran plus loin, parce que rien n'est téléchargé à moins qu'un bouton d'export n'existe et ne soit journalisé. Cette console est plus lente et moins agréable qu'un client natif, et je la proposerais comme option pour les personnes qui se contentent de regarder et n'ont jamais besoin de leurs propres outils, pas comme une obligation.

Décider explicitement de ce qui sort. La possibilité d'exporter ou non un résultat depuis une session est une règle à part entière, et elle relève du travail de masquage et de minimisation du Chapitre 12 et des chemins d'export du Chapitre 13. La gateway peut appliquer la règle, mais elle ne peut pas la rendre inutile. Une personne peut lire un écran, le prendre en photo ou ressaisir une valeur, et je ne connais pas de contrôle rentable contre cela. La gateway réduit la quantité de données qui sort par accident et rend les exports délibérés visibles ; elle ne rend pas la sortie impossible.

> **Remarque :** Si vos ingénieurs ont besoin de gros extraits pour des analyses, la réponse n'est généralement pas une limite d'export plus élevée dans la gateway. C'est une copie d'analytics masquée conçue à cet effet, que décrivent les chapitres 6 et 12. La gateway est le mauvais outil pour déplacer des données en masse.

Une seule porte, un seul point de défaillance

Un point d'entrée unique est facile à expliquer, facile à journaliser et facile à fermer. C'est aussi un point de défaillance unique, et il vaut la peine de le dire clairement avant de le recommander. Si la gateway est en panne, personne ne peut atteindre la base de données par la porte principale, y compris l'ingénieur d'astreinte qui a le plus besoin de la base de données quand quelque chose d'autre est déjà cassé.

Aucune réponse ne supprime ce coût. Certaines réponses le déplacent là où vous pouvez le gérer.

Garder la gateway petite. Elle authentifie, crée des logins, transmet et enregistre. Chaque fonctionnalité supplémentaire est une raison de plus pour qu'elle tombe en panne. Je résisterais à l'idée d'ajouter la transformation des résultats et une interface d'approbation au même service, et je les ferais tourner comme des composants séparés que la gateway appelle mais dont elle ne dépend pas pour démarrer une session de lecture. Là où une instruction est vérifiée avant de s'exécuter, cette vérification est elle aussi un composant à part (Chapitre 9).

Faire tourner plus d'une instance, et mettre l'état dans un endroit durable. Deux instances derrière la même adresse, partageant les enregistrements de session, signifient qu'un redémarrage ou un mauvais déploiement ne met pas fin à toutes les sessions. Le stockage des sessions et le log d'audit sont aussi des dépendances, et leur disponibilité fait partie de la disponibilité de la gateway. C'est la raison pour laquelle la fonction d'ouverture ci-dessus refuse une session quand le log est injoignable : c'est un choix délibéré qui lie l'accès humain au log, et ce devrait être un choix que l'équipe a fait exprès et consigné.

Décider quels sont les modes dégradés. Il y a deux défaillances à prévoir. L'une est la gateway en panne alors que les bases de données vont bien. L'autre est le log d'audit en panne alors que la gateway fonctionne. Pour chacune, quelqu'un de responsable doit avoir décidé à l'avance si l'accès humain s'arrête ou continue par un chemin plus étroit, et ce qui est enregistré entre-temps. La décision du livre pour une panne de la gateway est qu'il n'y a pas de seconde porte : l'accès humain s'arrête, break-glass compris, puisqu'une session break-glass est une session de la gateway avec un rôle plus lourd. La façon de rendre cela rare, c'est la redondance dé-

crite ci-dessus, pas un login d'urgence en dehors de la gateway. Le Chapitre 8 donne le raisonnement et ce que fait une équipe si elle en décide autrement, et le Chapitre 15 décrit comment répéter la panne. Pour une panne du log d'audit, les lectures sur une réplique peuvent mettre les entrées en tampon, et les écritures ne le peuvent pas (voir les compromis ci-dessous).

Traiter la gateway comme de la production. Elle a un responsable, un chemin d'astreinte, un health check qu'une personne peut lire, et un processus de déploiement avec un rollback. L'équipe plateforme en est responsable. Si la gateway est le projet annexe d'un seul ingénieur, le point de défaillance unique, c'est cet ingénieur.

La gateway concentre aussi un autre risque : une compromission de la gateway est une compromission de l'accès à tout ce qu'elle peut atteindre. C'est pourquoi son login `ledgerly_access_admin` est protégé comme décrit plus haut, et ce n'est pas une raison d'éviter la gateway : l'état initial a, à la même position, un mot de passe partagé copié chez tous les membres de l'équipe.

Le passage de Ledgerly de nombreux chemins à un seul

Une équipe dans la position de Ledgerly ne basculerait pas tout en un jour. Je suivrais les étapes dans un ordre qui garde un chemin fonctionnel disponible à chaque moment.

1. **Inventorier les chemins.** Listez chaque façon dont une personne peut atteindre aujourd'hui l'une des bases de données de Ledgerly : les routes VPN, le bastion, les entrées enregistrées des clients, les scripts qui se connectent avec le login admin. Le tour d'Alex dans l'ouverture était cette étape, et il devrait aboutir à une liste écrite que l'équipe plateforme tient à jour.
2. **Mettre en place la gateway à côté des anciens chemins.** Au début, elle crée les logins de session sur le primary, où les logins sont définis, leur accorde `human_readonly` et les connecte à `ledgerly_replica_2`, et rien d'autre. Les anciens chemins fonctionnent toujours. Quelques personnes utilisent la gateway pour leurs lectures quotidiennes et signalent ce qui est malcommode.

3. **Déplacer d'abord la lecture.** La lecture représente l'essentiel de ce que font les gens, et c'est elle qui comporte le moins de risque à essayer. Une fois les points malcommodes corrigés, fermez la réplique au VPN et supprimez les entrées enregistrées qui pointaient vers elle.
4. **Déplacer le primary en dernier.** L'accès direct au primary est ce sur quoi les gens s'appuient pendant un incident, je ne le fermerais donc qu'une fois que les voies JIT et break-glass du Chapitre 8 fonctionnent et ont été répétées. Le fermer plus tôt retirerait le chemin d'urgence avant que son remplaçant ne soit prêt.
5. **Supprimer l'ancien bastion et la portée du login admin.** Le bastion du prestataire est éteint. Le login admin partagé cesse d'être accepté depuis les réseaux humains. Les services gardent leurs propres logins jusqu'à ce que le travail du Chapitre 4 les remplace.
6. **Tester les refus.** Pour chaque ancien chemin, quelqu'un l'essaie et confirme qu'il échoue. Un chemin refusé que personne n'a testé est une hypothèse.

Le coût de la migration est surtout humain. Les gens doivent apprendre un nouveau parcours de connexion, et certains de leurs scripts vont casser parce qu'ils supposaient un login permanent. Prévoyez une période pendant laquelle les deux fonctionnent, et consignez la date à laquelle l'ancien chemin ferme. Sans date, l'ancien chemin reste ouvert, parce qu'il semble toujours moins coûteux de le laisser.

Compromis et coûts

Une conception plus petite peut suffire. Une gateway est un nouveau système sur le chemin critique, et si vous ne pouvez pas pourvoir son chemin d'astreinte et ses déploiements, une conception plus petite est un meilleur point de départ. Si quatre ingénieurs atteignent une seule base de données par un réseau privé, avec des logins personnels de courte durée émis par quelque chose que votre équipe fait déjà tourner, vous avez peut-être déjà l'essentiel de la valeur. Je consignerais ce qui me ferait changer d'avis : l'arrivée d'un prestataire, une deuxième base de données, une demande d'enregistrement des sessions, ou le premier incident où le log n'a pas pu dire qui l'a fait. La gateway est une réponse à ces conditions, pas une étape obligée.

Échouer en mode fermé a un prix. Refuser une session quand le log d'audit est injoignable garde l'enregistrement complet et fait du stockage du log une dépendance de chaque session humaine. Certaines équipes préfèrent ouvrir les sessions de lecture avec un tampon local et envoyer les entrées plus tard. C'est un choix défendable pour des sessions en lecture seule sur une réplique, et un choix plus faible pour des sessions qui écrivent. Si vous le faites, précisez à quelles sessions il s'applique, et surveillez le tampon.

L'enregistrement des instructions est une copie de données. L'enregistrement par fingerprint masque les littéraux, mais il nomme toujours des tables et des colonnes, et tout champ laissé non masqué, comme une note en texte libre ou un commentaire fourni par le client, peut porter une valeur. Traitez le log avec la classification de la chose la plus sensible qu'un enregistrement pourrait contenir, restreignez qui le lit, et fixez une durée de rétention dont quelqu'un est responsable.

Latence et fidélité. Une gateway dans le chemin des données ajoute un saut et peut ne pas prendre en charge chaque fonctionnalité de client ou chaque extension de protocole. Certains outils se comportent bizarrement à travers un intermédiaire : les très gros transferts, les curseurs de longue durée et les parcours d'authentification inhabituels sont les endroits habituels. Testez les outils que vos équipes utilisent réellement avant de promettre que l'ancien chemin fermera.

Quand ne pas utiliser de gateway. Un prototype à un seul service sans données client n'en a pas besoin. Un système dont le seul accès humain passe par une console managée qui fournit déjà l'identité, des credentials de courte durée et la journalisation n'a peut-être pas besoin d'une deuxième couche devant elle. Un bastion est une réponse raisonnable quand le vrai besoin est un endroit unique pour fermer la porte réseau et que le système protégé a sa propre authentification forte et personnelle. Dans tous ces cas, les mêmes questions s'appliquent : qui est la personne, combien de temps dure l'accès, et où se trouve l'enregistrement.

Erreurs courantes

- **Traiter le VPN comme la couche d'identité.** C'est tentant parce que le VPN sait déjà qui y est connecté et que tout le monde l'a. Il prouve

l'appartenance au réseau, pas quel login de base de données a été utilisé ni ce qu'il a fait. Gardez le VPN comme moyen d'atteindre la gateway et ajoutez une couche qui associe une personne à chaque session.

- **Placer un bastion devant la base de données et considérer le travail comme terminé.** Il met de l'ordre dans le réseau et laisse le login partagé et les instructions invisibles. Choisissez un bastion pour ce qu'il fait, une porte réseau unique que l'on peut fermer, et cherchez l'identité et l'enregistrement ailleurs.
- **Laisser l'ancien chemin ouvert pendant la migration.** Il semble prudent de garder les deux chemins pendant un moment, et ça l'est, mais sans date de fermeture l'ancien survit au nouveau. Fixez la date, annoncez-la, et testez ensuite que l'ancien chemin échoue.
- **Ouvrir une session dont personne n'impose la fin.** Une limite de session qui n'existe que dans la documentation est un grant permanent sous un nom plus long. Terminez les sessions à `ends_at`, fermez les connexions en plus de supprimer le login, et faites tourner un sweep qui trouve les sessions et les logins qui ne devraient pas exister.
- **Faire l'impasse sur la décision du mode dégradé.** Une équipe qui construit une gateway sans décider de ce qui se passe quand elle est en panne décidera sous pression, généralement en rouvrant l'ancien chemin partagé « juste une minute ». Décidez à l'avance de la voie d'urgence, consignez-la et répétez-la.
- **Tout enregistrer et ne rien lire.** Capturer chaque instruction donne une impression de rigueur. Sans un responsable qui l'échantillonne et une revue des accès qui la regarde, le log est une copie de données sans finalité. Décidez qui le lit et ce qu'il y cherche.
- **Lire la liste réseau comme plus précise qu'elle ne l'est.** Sur un réseau de cluster partagé, la règle ne peut pas distinguer un service d'un autre, et un relecteur qui suppose qu'elle le peut approuvera un chemin plus large qu'il n'y paraît. Consignez plutôt quel contrôle distingue les appelants.

Points clés

- La portée réseau et l'accès à la base de données sont des questions distinctes, et une conception qui répond aux deux avec le même contrôle ne peut pas dire quelle personne a fait quoi.

- Un bastion rend le chemin réseau unique mais ne comprend pas le protocole de la base de données, si bien qu'à lui seul il laisse le login partagé et les instructions invisibles.
- Une access gateway authentifie une personne par le single sign-on, crée un login de session qui porte cette identité et le supprime à la fin de la session, et c'est ce qui permet au log de la base de données de nommer une seule personne.
- La règle réseau qui n'autorise que la gateway à atteindre les hôtes de base de données est le contrôle qui fait de la gateway la seule porte, et elle devrait être une courte liste qu'une revue des accès peut lire.
- Un point d'entrée unique est un point de défaillance unique, donc la gateway devrait être petite, tourner en plus d'une instance, appartenir à l'équipe plateforme et s'accompagner d'une décision sur le mode dégradé prise avant l'incident.
- La gateway ne peut pas contrôler ce qu'une personne fait des données arrivées sur un ordinateur portable ; elle peut supprimer les credentials enregistrés, fixer des valeurs par défaut en lecture seule et rendre les exports visibles, et le reste relève du masquage et de la minimisation.
- Migrez d'abord la lecture, fermez le primary en dernier, fixez une date de fermeture pour chaque ancien chemin, et testez que les chemins refusés sont bien refusés.

La gateway donne à chaque personne une session qui porte un nom et une lecture par défaut, mais la lecture est rarement tout le travail. Le chapitre suivant examine ce qui se passe quand quelqu'un a besoin de plus que `human_readonly` pendant une durée limitée : l'accès just-in-time, l'accès break-glass, et ce que l'approbation coûte aux personnes qui l'attendent.

Chapitre 8 — Accès just-in-time et break-glass

Imaginez un mardi soir chez Ledgerly, un peu après trois heures du matin. Une alerte signale qu'un lot de factures est bloqué dans le statut `sending` depuis le redémarrage de `invoice-worker`, et les e-mails de relance du run du matin ne partiront pas tant que le lot n'aura pas été débloqué. Rien n'est exposé et rien n'est perdu ; le coût du problème est un retard. Robin, l'ingénieur backend d'astreinte, ouvre une session de routine via l'access gateway. La session est en lecture seule et s'exécute sur une réplique, ce qui est exactement ce que Robin veut pour les dix premières minutes : regarder, compter, comprendre.

Ce coup d'œil confirme le problème, et la correction exige un `UPDATE` sur une colonne de `invoices`. Robin rédige la correction d'une seule instruction et dépose une demande pour elle. L'approbateur est l'ingénieur d'astreinte secondaire, qui dort, et l'appel d'astreinte n'a pas été acquitté. Robin dispose d'un ordinateur portable, d'un runbook et d'une décision à prendre : attendre une réponse, ou trouver un moyen de contourner la question. Dans l'ancien Ledgerly, la décision n'aurait pas existé, parce que le mot de passe admin partagé était à un copier-coller. Dans le nouveau Ledgerly, le contournement doit être conçu, et le retour aussi.

Ce chapitre ajoute les étapes suivantes de l'état cible de l'exemple fil rouge : les humains n'ont aucun accès permanent en écriture, les écritures arrivent sous forme de grants just-in-time à l'intérieur du plafond du Chapitre 3, et il existe un rôle d'urgence rarement utilisé et toujours revu. Parmi les cinq parties du livre, il s'agit surtout du Grant : qui approuve quoi, pour quelle instruction, pour combien de temps. Il répond à la question de Robin : comment l'accès apparaît-il quand on en a besoin, comment se termine-t-il avec la tâche, et que se passe-t-il quand le chemin normal est trop lent ?

Le problème, et pourquoi il est difficile

L'accès permanent est la promesse que personne n'aura plus jamais à demander. C'est pratique pendant un mois, et ensuite plus personne ne se souvient de ce que couvrait la promesse. Les gens changent d'équipe et gardent leurs grants ; un rôle élargi pour un incident reste large. Quand arrive une revue des accès (Chapitre 14), le relecteur se retrouve face à des grants auxquels aucune tâche n'est rattachée, et la réponse à « est-ce encore nécessaire ? » est « probablement ». Le coût d'un grant permanent, c'est chaque jour où il ne sert pas, parce qu'il porte toujours le rayon d'impact de tout ce qu'il couvre.

Le remplaçant évident est l'**accès just-in-time (JIT)** : un accès accordé pour une tâche précise et pour une durée limitée. Il est facile à énoncer et facile à rater de façon subtile à cinq endroits.

Le périmètre. L'implémentation facile accorde un rôle à un login pour une heure. Si ce rôle est `human_jit_write`, le login reçoit le plafond entier, des droits de mise à jour sur six tables pour une tâche qui avait besoin d'une colonne d'une seule, et le grant est un grant permanent auquel on a attaché un minuteur. Le périmètre doit vivre dans les grants eux-mêmes, une demande à la fois.

L'expiration. Sur les moteurs courants, les grants n'expirent pas d'eux-mêmes. Quelque chose doit les reprendre, généralement un job planifié, et un job peut s'arrêter, perdre son credential, échouer sur un login et continuer, ou s'exécuter sur la mauvaise liste.

L'approbation. Une seconde personne qui lit chaque demande a un coût humain : quelqu'un est réveillé, quelqu'un attend. Si les approbations sont trop lentes, les gens trouvent le chemin plus rapide, qui est le mot de passe partagé que le livre cherche à retirer. Si l'approbateur répond toujours en quelques secondes, l'approbation ne réduit plus l'exposition.

L'urgence. Un dispositif d'accès sans moyen autorisé d'aller vite finit par en voir pousser un qui ne l'est pas. La question est de savoir si le chemin rapide a été conçu, journalisé et revu, ou improvisé.

L'opération. Un grant nomme une table et une colonne et ne dit rien des lignes ni de l'instruction. Un approbateur qui lit « `UPDATE` sur `invoices`, `status` » a approuvé une capacité, et tout ce qui s'exécute à l'intérieur

échappe à l'approbation. Pour que l'approbation veuille dire « une seconde personne a vu ce qui va se passer », la demande doit nommer l'opération et le grant doit en découler.

Une politique qui dit « l'accès doit être temporaire » ne dit pas au système quoi faire quand le job de révocation est en panne. La suite du chapitre relie les grants, les enregistrements et les minuteurs pour qu'il le sache.

L'accès just-in-time: des demandes qui nomment une opération

Une demande JIT se construit à partir de cinq éléments, chacun présent à cause de l'une des défaillances ci-dessus. Chacun porte l'étiquette de la partie du modèle du livre qu'il sert.

- Le plafond (Grant), `human_jit_write`, définit le maximum qu'une demande peut solliciter. Il n'est jamais accordé à un login (Chapitre 3).
- Une demande d'écriture (Grant) nomme la correction revue à laquelle elle se rapporte, dans `fix_id`. Ses triplets (privilège, table, colonnes) sont dérivés de cette correction, à l'intérieur du plafond, et stockés dans `access_request_privilege`.
- L'approbation (Grant) porte sur l'instruction, et le grant en fait respecter les bornes : l'approbateur lit l'instruction, la gateway accorde exactement les triplets dérivés au login de session du demandeur, et l'ensemble d'instructions approuvé est enregistré sous la forme `approved_hash`. L'expiration révoque exactement ces triplets, lus dans le même enregistrement.
- Dans une session JIT, les écritures (Exécution) ne sont transmises que depuis le runner, qui refuse tout ce qui ne correspond pas à `approved_hash`.
- Une réconciliation indépendante (Enregistrement) compare les grants que détient la base de données aux demandes qui les justifient, si bien qu'un job arrêté est remarqué par autre chose que le job lui-même.

Je les prendrai dans l'ordre où une demande les rencontre.

Ce que contient une demande

Une demande se fait depuis la session qui l'utilisera, et cette session s'exécute sur le primary. Une session porte au plus une demande active ; une tâche qui dépasse sa fenêtre continue dans une nouvelle session avec une nouvelle demande. L'investigation de routine reste sur la réplique avec `human_readonly` (Chapitre 6), et le Chapitre 7 dit qui peut ouvrir une session dirigée vers le primary. La gateway sait quel login appartient à quelle personne, si bien que personne ne saisit de principal ni de login.

Voici la forme de la demande de Robin, avant que quiconque y ait répondu :

```
access_request:
    id:              "req-0412"
    kind:            "jit"
    requester:       "robin"
    requester_login: "sess-robin-0412"
    approver:        null
    fix_id:          "fix-0093"
    approved_hash:   null            // set at approval, never by the
runner
    reason:          "alert-7731: invoices stuck in sending"
    expires_at:      now() + 1 hour        // illustrative window
    status:          "pending"

access_request_privilege:            // derived from fix-0093, not
typed
    ("UPDATE", "invoices", "status")
```

Trois choses méritent d'être remarquées. Le motif renvoie à quelque chose de vérifiable, une alerte ou un ticket. Le grant est étroit, et Robin n'a besoin d'aucun triplet `SELECT` parce que la session détient déjà `human_readonly`. Et Robin n'a pas saisi le triplet : il a été dérivé de `fix-0093`, si bien que le grant et l'instruction que lit l'approbateur ne peuvent pas diverger. Une demande qui ne contient que des triplets `SELECT`, par exemple pour lire une colonne Confidential non masquée, n'a ni correction ni hash.

Un grant restreint les privilèges et les colonnes, pas les lignes. L'`UPDATE` ci-dessus peut modifier `status` sur toutes les factures que la session peut voir. Un grant JIT borne ce qu'une erreur peut atteindre par colonne et

dans le temps ; ce qui la borne par ligne, c'est l'instruction que l'approbateur a lue et que le runner exécute, avec la discipline du comptage préalable du Chapitre 10.

Lier la demande à l'instruction

La correction est un ensemble d'instructions revu avec son filtre et son nombre de lignes attendu, sous la forme que décrit le Chapitre 10, et la demande la nomme par `fix_id` et porte son `approved_hash`. Le hash provient de l'enregistrement de merge de la correction ou de l'enregistrement de la demande approuvée, et jamais de l'entrée du runner ; dans une session JIT d'écriture, la gateway ne transmet les écritures que depuis le runner. Une demande qui ne contient que des triplets `SELECT` n'a ni correction, ni hash, ni runner, et l'approbateur lit la liste des colonnes. Le Chapitre 9 énonce la règle complète et explique comment la gateway vérifie chaque instruction.

Vérifier une demande

La gateway vérifie la forme de la demande avant que quiconque soit sollicité pour l'approuver, et la vérifie de nouveau, avec l'approbateur, au moment du grant.

```
function checkRequestShape(request):
    // runs before anyone is asked to approve
    // live = approved and not yet revoked; this request is not yet
counted
    if request.status != "pending" or
hasLiveRequest(request.requester_login):
        raise GrantRefused
    if request.expires_at is null or request.expires_at <= now():
        raise GrantRefused
    // a window longer than the gateway allows is refused, not clamped
    if request.expires_at > now() + maxJitWindow:
        raise GrantRefused
    for each p in request.privileges:
        // an empty column list is never read as "all columns"; table-
level
        // triples exist only on engines using the secrets-tables
fallback
        if p.privilege != "DELETE" and isEmpty(p.columns):
            if not tableLevelAllowed(p.table):
                raise GrantRefused
    // refuses a table outside the ceiling and any Restricted column
    if not withinCeiling("human_jit_write", request.privileges):
        raise GrantRefused
    checkOperationBinding(request)
```

```
function checkOperationBinding(request):
    if hasWriteTriple(request.privileges):
        fix = fixRecord(request.fix_id)    // null when fix_id is null
        if fix is null:
            raise GrantRefused
        // the triples come from the fix; a typed list that differs is
refused
        if request.privileges != triplesFromFix(fix):
            raise GrantRefused
    else if request.fix_id is not null or request.approved_hash is not
null:
        // a SELECT-only request has no fix and no hash
        raise GrantRefused

function checkJitRequest(request):
    // runs again at grant time, once an approver is named
    checkRequestShape(request)
    if request.approver is null or request.approver ==
request.requester:
        raise SelfApprovalNotAllowed
    if request.fix_id is not null:
        if request.approver == fixRecord(request.fix_id).author:
            raise SelfApprovalNotAllowed
```

Les lignes sur l'auto-approbation sont l'endroit où le principe des quatre yeux est appliqué dans le code : la personne qui demande, ou qui a écrit la correction, ne peut pas être celle qui répond. Une demande en attente n'a pas d'approbateur, c'est pourquoi la vérification de forme laisse ces lignes de côté ; l'approbateur est renseigné au moment de l'approbation, et la vérification complète s'exécute au moment du grant.

Je refuserais une fenêtre trop longue plutôt que de la raccourcir en silence : une fenêtre raccourcie laisse le demandeur croire qu'il dispose d'une heure alors qu'il a vingt minutes, et la surprise arrive en pleine réparation.

`withinCeiling` est l'endroit où le plafond du Chapitre 3 devient applicable. Il rejette un triplet dont la table est hors du plafond, dont le privilège n'est pas proposé sur cette table, ou dont la liste de colonnes contient `iban`, `account_holder_name` ou `token_value`. Il rejette `INSERT` et `DELETE` sur `bank_accounts` et `payment_tokens`, parce qu'une ligne ajoutée ou supprimée crée ou détruit des valeurs Restricted. Il rejette tout triplet autre qu'un `DELETE` qui n'a pas de liste de colonnes, sauf un triplet au niveau de la table sur un moteur qui utilise le fallback des tables de secrets. Et il rejette tout ce qui touche `ledger_entries`, `audit_events` ou les enregistrements des demandes d'accès, qu'aucun rôle humain n'écrit.

Une chose reste volontairement à l'intérieur du plafond. `UPDATE (customer_id)` sur `bank_accounts` ou `payment_tokens` passe `withinCeiling`, parce que `customer_id` ne fait que relier une ligne à son propriétaire et n'est pas Restricted. Déplacer la ligne vers un autre client change malgré tout à qui appartient le compte ou le token, si bien que la politique d'exécution l'oriente vers le chemin produit (Chapitre 9). La checklist de l'approbateur le détecte à la main.

Qui approuve, pour combien de temps, avec quel périmètre

Une demande JIT a trois variables : l'approbateur, la fenêtre et le périmètre.

L'approbateur est une personne qui comprend les tables de la demande et qui n'est ni le demandeur ni l'auteur de la correction. Chez Ledgerly, l'approbateur naturel pour `invoices` est un autre ingénieur backend, généralement l'ingénieur d'astreinte secondaire. L'approbateur naturel pour tout ce qui concerne `payments` ou `payment_tokens` est quelqu'un de l'équipe

responsable de `payments-service`, parce que cette équipe sait ce qu'un `status` modifié à cet endroit déclenchera en aval. L'équipe plateforme exploite l'outillage d'accès, et je n'en ferais pas l'approbateur par défaut pour des données dont elle n'est pas responsable : elle devient le goulot d'étranglement, et un approbateur incapable de juger le changement.

```
function chooseApproverPool(request):
    owners = emptySet()
    for each p in request.privileges:
        owners.add(owningTeam(p.table))
    pool = eligibleApprovers(owners)
    // neither the requester nor the author of the fix may answer
    pool.remove(request.requester)
    if request.fix_id is not null:
        pool.remove(fixRecord(request.fix_id).author)
    if pool is empty:
        raise NoApproverAvailable
    return pool
```

Un pool vide échoue de façon visible et ne se rabat jamais sur un pool plus large. Sinon, l'ordre du pool est l'ordre d'escalade : l'appel d'astreinte va à la première personne, puis à la suivante, au sein de la même demande en attente, qui n'est refusée que lorsqu'elle atteint `expires_at` sans réponse.

La fenêtre doit être aussi longue que la tâche typique, pas aussi longue que la pire. Disons qu'une correction de données prend généralement vingt minutes, que le formulaire propose une heure et que le maximum de la gateway est de quatre heures. Prolonger un grant, c'est faire une nouvelle demande avec une nouvelle approbation, pas modifier l'ancienne, parce qu'une modification permettrait à une seule approbation de s'étirer indéfiniment. Quand une réparation dépasse sa fenêtre, la session se termine et toute transaction ouverte fait un rollback avec elle, si bien qu'une réparation écrite sous forme d'une seule transaction (Chapitre 10) perd son travail proprement au lieu d'en laisser la moitié appliquée.

Le périmètre, ce sont les triplets et l'instruction dont ils proviennent. L'approbateur lit les deux, et l'écran d'approbation montre les triplets sous la forme des grants qu'ils deviendront : « UPDATE sur `invoices` (`status`), jusqu'à 04:12 ». Une demande qui nomme une colonne Confidential dans un `SELECT` montre cette colonne non masquée, et l'écran le dit en toutes lettres, pas seulement dans la liste des colonnes.

L'approbation a lieu sur l'écran propre de la gateway, via single sign-on avec une nouvelle vérification multifacteur au moment du clic, et non comme une réaction sous un message de chat : l'approbation est le contrôle, donc ce qui l'enregistre doit savoir qui l'a donnée.

Ce que vérifie l'approbateur

La décision de l'approbateur est plus étroite que « est-ce une bonne idée ? » : approuver l'instruction, et laisser le grant en faire respecter les bornes. Cette checklist reflète, du côté de l'accès, la checklist du relecteur pour la correction au Chapitre 10.

1. Le motif est-il une alerte ou un ticket vérifiable, et est-il toujours ouvert ?
2. L'instruction affichée à l'écran fait-elle ce que dit le motif ? Lisez l'instruction, pas le résumé qu'en fait le demandeur.
3. Le filtre borne-t-il les lignes, et le nombre attendu correspond-il à ce que le demandeur a vu ?
4. Les triplets dérivés couvrent-ils l'instruction et rien de plus ? Sous le fallback des tables de secrets, « rien de plus » signifie « toutes les colonnes non Restricted de la table », et l'écran le dit.
5. La fenêtre est-elle à peu près aussi longue que cette tâche, et sous le maximum ?
6. L'instruction touche-t-elle `customer_id` sur `bank_accounts` ou `payment_tokens`, ou une colonne Restricted ? Alors ce n'est pas une correction de données JIT. Envoyez-la vers le chemin produit et refusez la demande.
7. Ne suis-je ni le demandeur ni l'auteur, et serais-je à l'aise pour expliquer cette approbation au relecteur la semaine prochaine ?

Un approbateur qui ne peut pas répondre à l'une de ces questions interroge le demandeur pendant que la demande reste en attente. Le silence n'est pas une approbation, et une liste de colonnes non plus.

L'octroi

L'approbation est le moment où quatre enregistrements doivent concorder : l'enregistrement de la demande dans la gateway, l'entrée du log d'audit, les grants dans la base de données et la session. Aucune transaction ne

les couvre tous, donc l'ordre compte, tout comme ce que chaque défaillance laisse derrière elle.

```
function grantJitAccess(request):
    checkJitRequest(request)

    // the hash comes from the records, as "Binding the request"
explains
    hash = approvedHashFor(request)
    // the approved record comes first: the revoke job reads it, so a
grant
    // can never exist without its triples and its end
    markApproved(request.id, approved_hash: hash)

    // audit next: if the log is unreachable, no grant is made
    try:
        auditLogAppend({ actor: request.approver, action: "grant",
                         subject: request.requester_login,
                         request: request.id, fix: request.fix_id,
                         approved_hash: hash, triples:
request.privileges,
                         expires_at: request.expires_at })
    on AuditLogUnavailable:
        markRefused(request.id)
        raise GrantRefused

    // the session's end follows the approved window, as "Ending the
grant" says
    setSessionEnd(request.requester_login, request.expires_at)

    // separate systems, no shared transaction: a failure part-way
leaves an
    // approved record with no grant or only some triples
    for each p in request.privileges:
        grant(p.privilege, on: p.table, columns: p.columns,
              to: request.requester_login)

    // the window may end between the check and the grants: end it now
    if request.expires_at <= now():
        revokeRequestAccess(request.id, from: request.requester_login)
```

Lisez-le à travers ses modes de défaillance. Si le log d'audit est en panne, la demande est refusée et rien n'est accordé : un grant qui ne peut pas être enregistré ne peut pas être revu, et je préfère arrêter une réparation plutôt que d'en créer une invisible. Si la gateway tombe après l'enregistrement et l'entrée de log mais avant les grants, la demande est approuvée et reste sans effet, ce qui n'expose rien. Si elle tombe après deux triplets sur trois,

le login en détient deux, l'enregistrement en contient trois, et la révocation ultérieure reprend les trois, puisque révoquer un triplet qui n'a jamais été accordé est sans danger.

Les dernières lignes traitent une situation de concurrence. La fenêtre ne redémarre pas à l'approbation, donc la vérification de forme refuse une approbation qui arrive après `expires_at` ; il reste le cas d'une approbation qui passe un instant avant la fermeture de la fenêtre, et ces lignes reprennent ses grants immédiatement.

L'entrée d'audit porte la correction et le hash en plus des triplets, si bien que l'approbateur, l'instruction et la fenêtre se trouvent dans un seul enregistrement. Les régimes applicables aux données de paiement et au reporting financier ont tendance à demander qui a approuvé une modification des données de production, sur quelle base et pour combien de temps, et cet enregistrement répond aux trois ; l'interprétation revient aux équipes juridique et conformité.

Mettre fin au grant

Un grant prend fin en trois couches, et aucune d'elles n'est considérée comme la seule.

La première couche est la session. La gateway y met fin à `expires_at` et supprime le login, sans aucun job planifié : l'horloge qui borne l'accès appartient au composant qui l'a créé. Une session dirigée vers le primary s'ouvre avec le plafond de deux heures du Chapitre 7 ; lors d'un grant, depuis `grantJitAccess` ou `declareBreakGlass`, la gateway ramène donc le `ends_at` de la session au `expires_at` de la demande, et le `endSession` du Chapitre 7 ferme la session à ce moment-là ; une fenêtre de plus de deux heures n'est donc pas écourtée.

```
function endSessionAtExpiry(session):
    // one live request per session, so its end is the session's end
    waitUntil(session.request.expires_at)
    // never raises; a failure leaves the record for the next sweep
    endSession(session.login, "expired")
```

La fin elle-même, c'est `endSession`, l'unique endroit qui ferme les connexions, révoque ce que les demandes de la session ont accordé et supprime le login, et `sweepSessions` appelle la même fonction (Chapitre 7).

La deuxième couche est le job de révocation, qui lit le même enregistrement et retire ce que la demande a accordé. `revokeRequestAccess` dérive son travail de `access_request_privilege` : pour une demande JIT, il révoque chaque triplet stocké ; pour une demande break-glass, il révoque le rôle `human_breakglass`. Il ne lit pas une liste séparée qui pourrait diverger, et il traite un login manquant comme déjà révoqué.

```
function revokeExpiredGrants():
    // expiredGrants: kind jit or breakglass, status approved,
expires_at
    // passed, not yet revoked
    for each item in expiredGrants(before: now()):
        try:
            // revokes what the request granted: its stored triples
for "jit",
            // the role human_breakglass for "breakglass"
            revokeRequestAccess(item.request_id, from: item.login)
        on LoginNotFound:
            // the login went with the session; nothing is left to
revoke
        on RevokeFailed:
            // an expired grant still live is an incident; retried
next run
            notify("platform", "expired grant still active: " +
item.id)
            continue
        markRevoked(item.request_id)
        try:
            auditLogAppend({ action: "revoke", actor: "revoke-job",
                             subject: item.login, request:
item.request_id })
        on AuditLogUnavailable:
            // the revoke is done; the log catches up from the retry
queue
            queueForRetry(item.request_id)
            notify("platform", "revoke not yet logged: " + item.id)
```

La révocation continue quand le log d'audit est en panne. L'asymétrie est délibérée : un grant sans entrée de log est une exposition, alors qu'une révocation sans entrée est un manque qui peut être comblé plus tard.

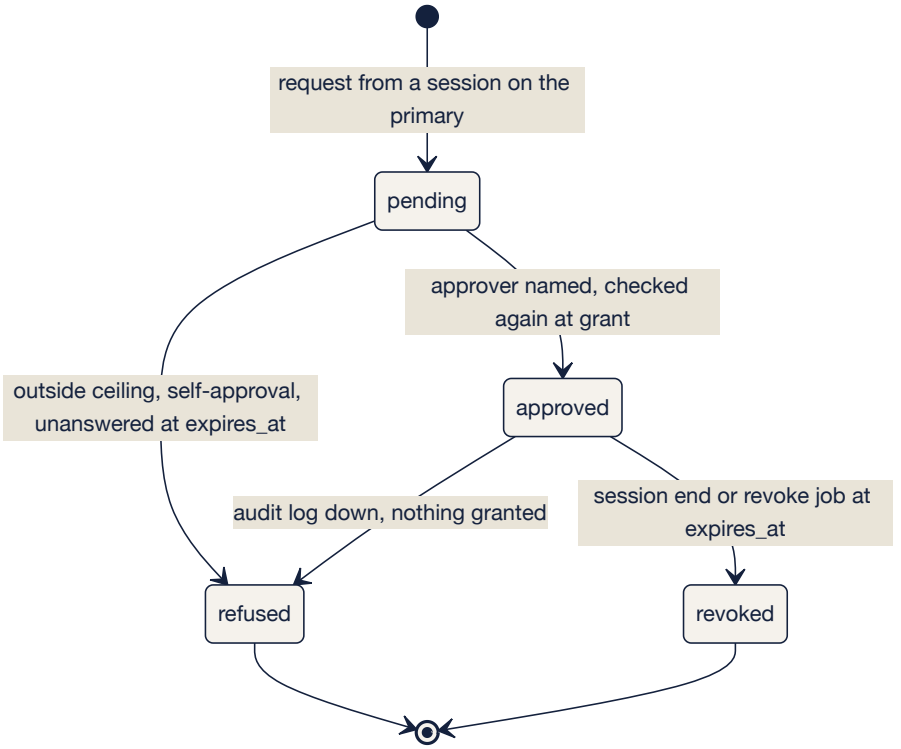
La troisième couche est la réconciliation, et elle est d'une autre nature. Elle ne retire pas d'accès. Elle demande, depuis l'extérieur, si les deux premières couches ont fait leur travail :

```
function checkOverdueGrants():
    // scheduled separately from revokeExpiredGrants, so one failure
    // cannot hide the other
    if count(expiredGrants(before: now() - graceMinutes)) > 0:
        notify("platform", "grants past expiry and not revoked")
    if now() - lastSuccessfulRun("revokeExpiredGrants") > maxRunAge:
        notify("platform", "revoke job has not completed recently")
```

À ses côtés, la vérification de dérive du Chapitre 3 lit les grants que la base de données détient réellement et les compare aux triplets non expirés des demandes. Un grant direct sans demande active derrière lui, un login membre du plafond, un membre de `human_breakglass` sans demande break-glass ouverte : chacun est signalé, même quand le job de révocation annonce un succès. La vérification des grants échus surveille les enregistrements sur lesquels le job aurait dû agir ; la vérification de dérive surveille la base de données que le job aurait dû modifier. Un job qui annonce « terminé » sans rien modifier n'est détecté que par la seconde.

La réconciliation ne répare rien d'elle-même, et je la garderais ainsi : une révocation automatique de grants inconnus est un moyen de casser la production en pleine nuit. Elle notifie l'équipe plateforme, et une personne décide.

Les états que traverse une demande sont peu nombreux, et les dessiner montre où agit chaque couche. Regardez les sorties de `approved` : une révocation, ou un refus quand le log d'audit est en panne et que rien n'a été accordé ; rien ne revient de `revoked`.



Auto-approbation, et pas d'approbation automatique pour l'astreinte

L'auto-approbation est refusée, toujours. `checkJitRequest` lève `SelfApprovalNotAllowed` et rien ne passe outre, ni l'ancienneté ni l'urgence. Le cas urgent a un autre chemin, qui fait l'objet de la seconde moitié de ce chapitre.

L'approbation automatique est l'autre raccourci tentant, parce que la friction est la plus forte la nuit. Pour les lectures, je ne la construirais pas : chaque login de session détient déjà `human_readonly`, et la seule chose qu'ajoute un `SELECT` JIT est une colonne Confidential affichée non masquée, ce qui est exactement ce qu'un approbateur devrait regarder. Une écriture change ce sur quoi les services agiront ensuite, et une machine ne peut pas dire si `status` est la bonne colonne. Gardez les écritures avec un approbateur humain et consacrez l'effort à rendre cet approbateur facile à joindre.

Si vous ajoutez plus tard une approbation automatique, pour une lecture qui apporte réellement quelque chose de nouveau, la règle doit exiger une liste de colonnes explicite, et une liste vide ou absente doit faire refuser la demande de façon visible. Sous le fallback des tables de secrets, une lecture au niveau de la table n'a aucune liste de colonnes, si bien qu'une règle qui considère « aucune colonne Confidential nommée » comme acceptable approuverait toutes les colonnes Confidential de la table.

Les demandes issues de l'automatisation, et la politique sous forme de code

Un job, un script de runbook ou un agent d'IA qui veut écrire est un principal comme un autre : il ouvre sa session sous sa propre workload identity, nomme une correction et attend un approbateur humain, et l'enregistrement nomme à la fois l'automatisation et la personne qui l'a déclenchée (le Chapitre 9 couvre ce qu'il peut faire à l'intérieur d'une session). La logique d'approbation est elle-même du code. `withinCeiling`, `chooseApproverPool`, `maxJitWindow` et les vérifications de forme vivent dans le dépôt, passent en revue et font partie de la baseline de la vérification de dérive ; un plafond modifié à la main dans une console, c'est le problème d'origine un niveau plus haut.

Cas d'abus

Trois cas sont bornés par les contrôles ci-dessus et éliminés par aucun. La collusion entre demandeur et approbateur est restreinte aux colonnes, à l'instruction et à la fenêtre de l'enregistrement, et le binôme est visible dans l'entrée d'audit ; c'est dans la revue des accès (Chapitre 14) qu'apparaît un binôme qui s'approuve mutuellement bien plus souvent que le hasard ne le voudrait. Un approbateur manipulé par ingénierie sociale (« approuve req-0412 tout de suite, le lot bloque la paie ») se heurte à la checklist : lire l'instruction sur l'écran de la gateway, vérifier l'alerte indiquée dans le motif, et ne répondre à rien de ce qui arrive par un autre canal. Un token de single sign-on détourné ne peut pas écrire dans une session JIT, parce que la gateway ne transmet les écritures que depuis le runner et que le runner refuse un hash qui n'a jamais été approuvé. Le risque résiduel se situe dans les lectures à l'intérieur des triplets accordés et dans le break-glass, qui n'a pas d'approbateur ; ce sont la vérification renforcée à la déclaration et l'alerte qui le bornent.

L'accès break-glass

Le **break-glass** est un accès d'urgence avec un ensemble de journalisation défini et une revue obligatoire. Le nom vient du déclencheur d'incendie derrière une vitre : à la portée de quiconque en a besoin, et le briser est visible de tous.

Dans le modèle du livre, c'est le rôle `human_breakglass`, et c'est le seul rôle humain qui lit les colonnes Restricted. Il lit toutes les tables et écrit `INSERT` et `UPDATE` sur toutes les tables sauf les deux tables append-only ; sur `bank_accounts` et `payment_tokens`, il n'écrit que `UPDATE`, et seulement sur les colonnes non Restricted. Il n'écrit jamais `ledger_entries` ni `audit_events`, parce qu'aucun rôle humain ne le fait. Ce n'est pas une voie pour les corrections de routine, et ce n'est pas un chemin de correction de données : une correction de données est planifiée, revue et bornée (Chapitre 10), et le break-glass n'est rien de tout cela, par conception.

Une demande break-glass est auto-déclarée. Elle n'a pas d'approbateur. La personne dans la session déclare qu'une urgence existe, donne un motif, et la gateway marque la demande comme approuvée et accorde le rôle au même login de session jusqu'à `expires_at`. Le même job de révocation et le même minuteur de session le reprennent.

L'approbateur constitue à lui seul le contrôle du JIT, si bien qu'un rôle d'urgence sans approbateur peut sembler étrange. Le break-glass existe pour les cas où le pool d'approbateurs n'a pas répondu, et une étape d'approbation reproduirait cette défaillance. Le contrôle passe d'avant l'accès à autour de lui : une alerte à la déclaration, l'ensemble de journalisation ci-dessous, une fenêtre courte et fixe, et une revue qui suit sans exception.

```
function declareBreakGlass(session, reason):
    // a hijacked token alone must not be enough to declare
    requireFreshMultiFactor(session.principal)
    // one live request per session: a break-glass declaration
supersedes it
    supersedeLiveRequest(session.login)
    // the window is the gateway's fixed maximum; there is no argument
for it
    window = breakGlassMaxWindow
    request = newRequest(kind: "breakglass", requester:
session.principal,
                         requester_login: session.login, approver:
null,
                         reason: reason, expires_at: now() + window)
    markApproved(request.id)
    try:
        auditLogAppend({ actor: session.principal, action:
"breakglass",
                         subject: session.login, request: request.id,
                         expires_at: request.expires_at })
    on AuditLogUnavailable:
        markRefused(request.id)
        raise GrantRefused
    setSessionEnd(session.login, request.expires_at)
    grant(human_breakglass, to: session.login)
    // paged after the grant: a paging outage must not block an
emergency;
    // the review queue below is the net for a page that never arrived
    queueReview(request.id, reviewer: "not-the-requester")
    notify("security", "break-glass declared: " + request.id)
    notify("platform", "break-glass declared: " + request.id)
```

Trois décisions de ce bloc méritent un commentaire.

La fenêtre est fixée par la gateway, à deux heures dans l'illustration de Ledgerly. Aucune fenêtre plus longue n'est disponible sur demande, parce que « j'en ai besoin un peu plus longtemps » transforme une exception en habitude. Si l'urgence dépasse la fenêtre, la personne déclare de nouveau et la seconde déclaration déclenche de nouveau l'alerte ; c'est la friction voulue.

La vérification du log d'audit est la même que pour le JIT. Si le log est en panne, la déclaration est refusée. J'accepte ce coût, parce qu'une session break-glass sans enregistrement est la seule chose qu'une revue ne peut pas reconstituer, et c'est la revue qui rend le rôle tolérable. Ce que fait l'astreinte quand le chemin normal et le log sont tous deux en panne relève du Chapitre 15.

L'alerte est une notification aux contacts sécurité et à l'équipe plateforme indiquant que quelqu'un a ouvert l'accès le plus large dont dispose Ledgerly, et non une demande de permission. Dans un système sain, elle est assez rare pour être lue ; si elle se déclenche chaque semaine, elle cessera d'être lue, et le break-glass sert alors à quelque chose qui mérite un chemin conçu.

L'ensemble de journalisation du break-glass

Une « journalisation renforcée » n'est pas une norme tant qu'elle n'est pas définie ; pour une session break-glass, l'ensemble de journalisation se compose donc de ces quatre éléments :

- Chaque instruction, sous forme de fingerprint dont les littéraux font l'objet d'une redaction et avec un hash des paramètres, avec le début, la fin et les lignes affectées. La redaction compte ici plus qu'ailleurs : cette session lit des colonnes Restricted, et un littéral dans le log les copierait à un endroit dépourvu de protection au niveau des colonnes.
- Les entrées au fil de l'eau. Chaque instruction est ajoutée au log d'audit pendant son exécution, et non collectée à la fin, si bien qu'une session interrompue laisse tout de même son enregistrement.
- Une note d'une ligne pour chaque écriture. La gateway refuse une écriture qui n'en a pas, ce qui coûte quelques secondes et donne au relecteur l'intention à côté de l'instruction.
- Le même enregistrement que pour toute autre session : le login, la personne, la demande et le motif, afin que les trois enregistrements du Chapitre 14 puissent être joints.

Un risque demeure, et il diffère selon l'état. Aujourd'hui, sans gateway ou avec une gateway qui se contente de fournir le credential, une session break-glass est le seul endroit où un `UPDATE` non borné sur une table qu'elle peut écrire reste possible. La fenêtre, les nombres de lignes, les entrées et la revue bornent les dégâts et les exposent ; ils ne les empêchent pas. Dans l'état cible, la gateway du Chapitre 9 exécute les sessions d'écriture avec l'auto-commit désactivé et un row cap, sa matrice refuse une écriture non bornée dans cette lane, et une écriture sans note est refusée. Ce qui reste possible, c'est une écriture qui tient sous le cap et touche quand même les mauvaises lignes. Le cap est ici le plus difficile à dimen-

sionner, parce qu'une urgence n'est par définition pas la tâche planifiée ; tant que la politique n'en fixe pas un, l'écriture non bornée est un risque accepté, et la raison d'être de la revue.

La revue après coup

Chaque usage du break-glass est revu, par une personne autre que celle qui l'a utilisé. La revue n'est pas une audience ; elle pose les questions que pose une revue de conception : s'agissait-il d'une urgence que le chemin normal ne pouvait pas servir ; qu'est-ce que la session a lu ou écrit ; a-t-elle lu des colonnes Restricted, et était-ce nécessaire ; et une demande JIT aurait-elle pu répondre au même besoin, et si oui, qu'est-ce qui aurait rendu cette demande assez rapide ?

C'est la dernière question qui rapporte. Chaque revue confirme que le break-glass était le bon outil, ou bien trouve dans le chemin normal une lacune qui en a fait le seul outil. Combler la lacune (un approbateur manquant, une fenêtre trop courte, une table que le plafond ne propose pas) réduit l'usage suivant. La revue a aussi besoin de sa propre horloge :

```
function checkBreakGlassReviews():
    for each r in breakGlassRequests(endedBefore: now()):
        review = reviewFor(r.id)
        if review is null and now() - r.expires_at > reviewDeadline:
            notify("security", "break-glass review overdue: " + r.id)
        if review is not null and review.reviewer == r.requester:
            notify("security", "break-glass reviewed by its own user:
" + r.id)
```

Une revue que personne ne fait est un contrôle qui a discrètement cessé d'exister ; son échéance déclenche donc une alerte.

Quand la gateway n'est pas là

Les humains n'atteignent la base de données que par l'access gateway (Chapitre 7), et les grants JIT comme break-glass sont accordés par la gateway à un login de session qu'elle a créé. Une panne de la gateway est donc aussi une panne du break-glass. C'est un compromis de la conception, et le Chapitre 7 énonce la même décision du point de vue de la gateway.

L'alternative est une seconde porte : un login d'urgence hors de la gateway. Elle est séduisante la nuit où l'on en a besoin et dangereuse toutes les autres nuits, parce qu'un credential qui fonctionne sans la gateway fonctionne sans sa journalisation, son identité et son expiration ; c'est le mot de passe admin partagé sous un meilleur nom.

Je préfère rendre la gateway difficile à perdre plutôt que de rendre son absence supportable en affaiblissant tout le reste : plus d'une instance, peu de dépendances (le log d'audit, les enregistrements des demandes), une défaillance répétée en exercice, et un signal de santé que l'astreinte voit en premier. Si toutes les instances sont en panne en même temps, il ne reste aucun chemin humain vers les données, et je n'en inventerais pas un ici. L'instrument scellé du Chapitre 3 existe pour le travail au niveau du moteur, il a besoin d'une route réseau que le Chapitre 7 ferme aux personnes, et il contourne les protections append-only et Restricted ; ce n'est donc pas un chemin d'accès aux données. Le Chapitre 15 revient sur ce qu'il faut faire dans ce cas.

Ce choix n'est pas gratuit. Une équipe dont la gateway est fragile peut raisonnablement vouloir un second chemin le temps qu'elle mûrisse. Écrivez alors ce qu'il est, qui peut récupérer son credential, comment son usage déclenche une alerte, et ce qui le retirera : un chemin avec un responsable et une date de fin est une décision, et un chemin sans eux est un reliquat.

Quand le moteur n'a pas de grants au niveau des colonnes

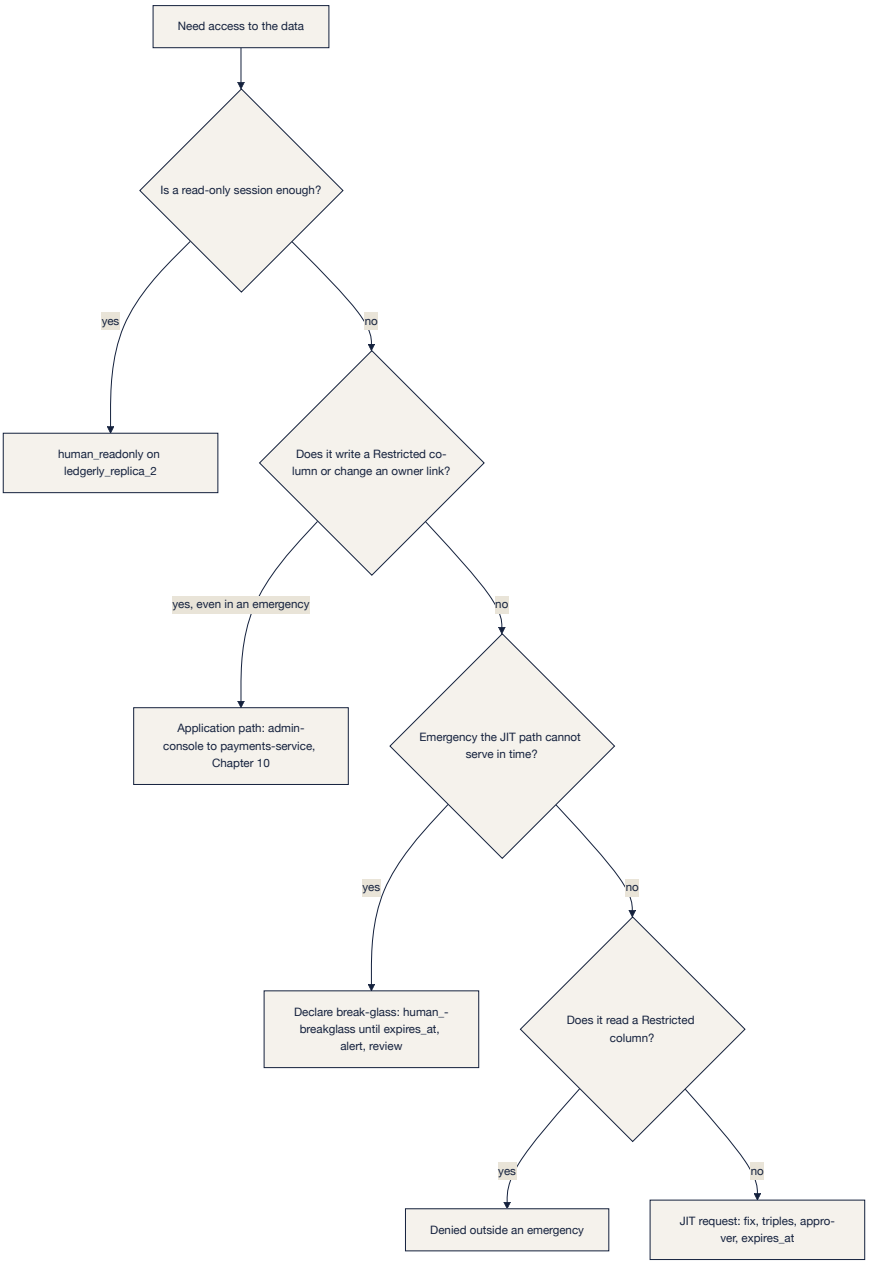
Tout ce qui précède suppose que le moteur peut accorder un privilège sur des colonnes nommées. Le Chapitre 3 a présenté l'unique fallback que le livre utilise pour les moteurs qui ne le peuvent pas : les colonnes Restricted vivent dans des tables séparées, `bank_account_secrets` et `payment_token_values`, hors du plafond et accessibles uniquement à `app_payments_service` et `human_breakglass`. Les tables principales ne contiennent alors aucune colonne Restricted, si bien qu'un grant JIT au niveau de la table sur `bank_accounts` ou `payment_tokens` ne peut pas en atteindre une. Une telle demande n'a pas de liste de colonnes : `columns` est null, et le triplet se lit « UPDATE sur `payment_tokens` ».

Le prix retombe sur l'approbateur, qui a approuvé toutes les colonnes non Restricted de la table, y compris les colonnes Confidential. Si le demandeur voulait modifier `expires_on`, l'approbateur approuve aussi `provider` et `last_four`. L'écran d'approbation doit dire « toutes les colonnes non Restricted de cette table » en toutes lettres, et l'instruction que lit l'approbateur porte une plus grande part du poids. Un groupe de colonnes qu'une équipe s'attend à ouvrir souvent mérite sa propre petite table.

Le fallback ne change pas les règles pour les données Restricted. Une personne ne les écrit toujours jamais en SQL brut ; la modification passe par `admin-console` et `payments-service` après l'approbation d'une seconde personne (Chapitre 10).

Choisir le chemin

Une personne à trois heures du matin a besoin de savoir quelle porte utiliser, et la réponse a sa place dans le runbook sous la forme d'un flux court. Le diagramme ci-dessous en donne la forme. Regardez les deux issues qu'aucun chemin humain vers les données ne sert : une écriture dans une colonne Restricted, qui va vers le chemin produit même en cas d'urgence, et une lecture d'une telle colonne en dehors d'une urgence, qui est refusée. La conception le dit au lieu de laisser la décision à quiconque se trouve sous pression.



Ledgerly à 3 h du matin

Retour à Robin. Les lignes bloquées ont été comptées dans la session de routine, et leur nombre correspond au runbook. L'écriture est la demande liée à `fix-0093`, pour une heure, qui cite l'alerte ouverte. `chooseApproverPool` sélectionne l'ingénieur d'astreinte secondaire et le

chef de l'équipe, dans cet ordre, et l'appel d'astreinte va au premier. L'ancien Ledgerly aurait ouvert le mot de passe partagé à ce stade. Le nouveau impose une attente, et cette attente, c'est la conception qui fonctionne.

Si personne ne répond, le runbook pose la question qui décide de tout : s'agit-il d'une urgence que le pool d'approbateurs ne peut pas servir à temps ? Des relances bloquées peuvent attendre le matin ; Robin laisse donc une note sur l'alerte et s'arrête, une issue que l'ancien mot de passe ne permettait jamais.

Supposons plutôt que `payments-service` soit défaillant, que l'encaissement des paiements soit arrêté pour tous les clients et que le diagnostic exige une colonne Restricted qu'aucune demande JIT ne peut contenir. C'est un cas pour le break-glass. Robin fait la déclaration avec l'alerte comme motif ; la gateway accorde `human_breakglass` pour sa fenêtre fixe ; les contacts sécurité et l'équipe plateforme sont prévenus. Le lendemain matin, une personne qui n'est pas Robin lit le log et pose les questions de la revue, dont l'une est de savoir pourquoi le diagnostic a eu besoin d'une colonne Restricted. La réponse pourrait déboucher sur une vue de diagnostic plus sûre au trimestre suivant.

Compromis et coûts

La friction est le prix, et il faut la dépenser là où elle achète quelque chose. Une demande JIT coûte au demandeur une minute pour l'écrire et à l'approbateur une minute pour la lire, dans le meilleur des cas, et un ingénieur réveillé dans le pire. Pour une tâche qui survient deux fois par trimestre, c'est bon marché. Pour une tâche qui survient deux fois par jour, c'est une fonctionnalité produit déguisée en opération, et le Chapitre 10 décrit comment la déplacer dans `admin-console`.

La liaison à une correction coûte de la préparation. La correction doit être écrite avant que la demande puisse être déposée, ce qui est plus lent qu'un simple grant. C'est le prix à payer pour approuver ce qui va s'exécuter plutôt que ce qui pourrait s'exécuter.

La fatigue d'approbation est une défaillance de la conception, pas de l'approbateur. Une approbation qui arrive en quelques secondes, à chaque fois, de la même personne, c'est du théâtre : soit les demandes sont triviales et `human_readonly` devrait les couvrir, soit l'approbateur a cessé de

lire. Le remède, ce sont des demandes plus petites, un écran qui montre ce que seront les grants, des approbateurs qui connaissent les tables et un pool assez grand pour que personne ne soit l'approbateur par défaut. Deux signaux ont leur place dans la revue des accès : la fréquence à laquelle les demandes sont refusées ou renvoyées, et la fréquence à laquelle c'est le même approbateur qui répond. Un taux de refus nul peut aussi bien signifier que personne ne regarde que signifier que les demandes sont bonnes.

C'est l'ingénieur d'astreinte qui porte le coût en premier. L'approbateur est réveillé, le demandeur attend, et l'équipe apprend un runbook qu'elle utilise rarement. Faites du pool d'approbateurs une rotation, publiez-la, et assurez-vous que les personnes qui y figurent ont le contexte nécessaire pour lire une demande concernant `invoices` ou `payments`.

La gateway est une dépendance de chaque écriture. Le job de révocation, le log d'audit et les enregistrements des demandes sont tous des dépendances d'un grant ; quand l'un d'eux est en panne, les grants s'arrêtent, ce qui limite l'exposition et ralentit la réponse aux incidents. Le Chapitre 7 décrit comment réduire les risques.

La réconciliation produit du bruit. La vérification de dérive et la vérification des grants échus signaleront des cas légitimes, comme une révocation qui s'exécute avec une minute de retard. Réglez le délai de grâce pour qu'un signalement veuille dire quelque chose ; une vérification mise en sourdine est pire que pas de vérification du tout, parce que l'équipe se croit couverte.

Le break-glass a un coût social. Une personne qui en déclare un dit aux contacts sécurité que le chemin normal a échoué. Si la revue ressemble à un procès, les gens hésiteront pendant les vraies urgences ; si c'est une revue de conception, ils n'hésiteront pas. La revue sans blâme est ce qui garde le rôle utilisable.

Quand ne pas utiliser cela. Une équipe de quatre ingénieurs avec une seule base de données à faible risque et aucune donnée Restricted n'a pas besoin de pools d'approbateurs ni d'un job de réconciliation dès le premier jour. Un processus manuel avec un motif, une limite de temps, une personne qui confirme et un enregistrement peut suffire pendant un temps, si

les grants sont retirés quand le temps est écoulé et que quelqu'un vérifie. Construisez le pattern dans l'ordre de ses défaillances : d'abord l'expiration, puis l'enregistrement, puis l'approbateur, puis l'alerte.

Erreurs courantes

- **Accorder le rôle plafond à un login « pour cette seule tâche ».** Chaque session JIT détient alors le plafond entier, et le périmètre de la demande est décoratif. Accordez les triplets au login de session et gardez `human_jit_write` comme un plafond auquel aucun login n'appartient ; la vérification de dérive signale un membre.
- **Approuver les triplets et non l'instruction.** Une liste de colonnes est courte et facile à lire, et l'approuver revient à approuver une capacité. Liez la demande d'écriture à une correction revue, dérivez-en les triplets, et prenez `approved_hash` dans l'enregistrement de la correction.
- **Révoquer à partir d'une liste séparée.** Une liste de tables codée en dur finira par oublier une table que la demande a accordée. Dérivez la révocation de `access_request_privilege`, l'enregistrement qui a piloté le grant.
- **Faire confiance au seul job de révocation.** Un job qui s'exécute chaque nuit depuis un an, on cesse facilement de le surveiller. Donnez à la session sa propre heure de fin et exécutez une vérification indépendante qui compare les grants aux demandes.
- **Écrire l'entrée d'audit après le grant.** Si le log est en panne, il existe un grant que personne ne peut voir. Écrivez l'enregistrement, puis l'entrée de log, puis les grants, et laissez une entrée de log ratée arrêter le grant.
- **L'approbation automatique des demandes de l'astreinte.** Les lectures n'ont pas besoin de demande et une écriture a besoin d'une seconde personne. Raccourcissez l'attente grâce à un bon pool d'approbateurs.
- **Rendre le break-glass assez long pour être confortable.** Une fenêtre qui n'expire jamais au cours d'une journée de travail devient un rôle de travail. Fixez le maximum dans la gateway et laissez une seconde déclaration déclencher de nouveau l'alerte.

- **Laisser l'ensemble de journalisation non défini.** Un relecteur ne peut pas juger une session dont l'enregistrement n'a jamais été spécifié. Nommez l'ensemble avant le premier usage.
- **Sauter la revue parce que rien ne s'est mal passé.** C'est dans les revues que le chemin normal se répare. Donnez une échéance à la revue et déclenchez une alerte quand elle est dépassée.
- **Construire une seconde porte pour les mauvais jours de la gateway.** Un login d'urgence hors de la gateway contourne son identité, sa journalisation et son expiration. Investissez d'abord dans la disponibilité ; si un second chemin est inévitable, donnez-lui un responsable, une alerte et une date de fin.

Points clés

- L'accès permanent coûte le plus cher les jours où il ne sert pas ; la question utile est donc de savoir qui a besoin d'accès maintenant, pour quelle tâche, et comment le système saura que la tâche est terminée.
- `human_jit_write` est un plafond et n'est jamais accordé à un login ; une demande d'écriture nomme sa correction revue (`fix_id`), les triplets sont dérivés de la correction, et l'approbateur approuve l'instruction, dont le hash (`approved_hash`) doit correspondre à celui du runner.
- La révocation est dérivée du même enregistrement `access_request_privilege` qui a piloté le grant, imposée par l'heure de fin propre à la session, soutenue par un job de révocation, et contrôlée par une réconciliation indépendante qui ne fait confiance à aucun des deux.
- Les écritures exigent le principe des quatre yeux : un approbateur qui n'est ni le demandeur ni l'auteur de la correction, et qui répond sur l'écran de la gateway après une nouvelle vérification multifacteur. Il n'y a pas d'approbation automatique pour l'astreinte.
- Le break-glass est une session `human_breakglass` auto-déclarée avec une fenêtre maximale fixe, un ensemble de journalisation défini, une alerte à la déclaration et une revue obligatoire par une personne autre que l'utilisateur. Ce n'est ni un chemin de routine ni un chemin de correction de données, et son unique risque accepté est, aujourd'hui, un

`UPDATE` non borné, et dans l'état cible, une fois que la gateway du Chapitre 9 impose un cap, de mauvaises lignes à l'intérieur du cap.

- Une conception sans chemin rapide autorisé en voit pousser un qui ne l'est pas ; la voie d'urgence est donc conçue, journalisée et répétée, et sa dépendance envers la gateway est un coût que vous nommez.
- Là où le moteur ne peut pas accorder de privilèges sur les colonnes, le fallback des tables de secrets tient les colonnes Restricted à l'écart de toute demande, et l'approbateur approuve des tables entières de colonnes non Restricted.

Un grant, même borné, dit ce qu'une session peut toucher et pour combien de temps, pas avec quel soin l'instruction qui se trouve derrière est vérifiée. Le Chapitre 9 aborde ce point : ce que la gateway fait d'une instruction avant qu'elle s'exécute. Le Chapitre 10 aborde ensuite le travail lui-même : comment exécuter une correction de données pour que les bonnes lignes changent, que les mauvaises ne changent pas, et qu'une seconde personne puisse voir pourquoi.

Chapitre 9 — Exécution médiée: analyse des requêtes et opérations dangereuses

L'équipe de support de Ledgerly dispose d'un assistant dans son outil de chat. Sam lui demande d'annuler un lot de factures en double issues d'un retry de facturation, et l'assistant, qui s'exécute sous sa propre identité, rédige une correction : un `UPDATE` sur `invoices` avec une clause `WHERE` portant sur le libellé de lot du retry. Jordan, de l'équipe data, relit l'instruction et l'approuve. L'instruction est du SQL valide, le grant la couvre, l'approbateur l'a lue, et le libellé était correct quand Jordan l'a vérifié.

L'instruction correspond pourtant à plus que les douze lignes que nommait la correction. Un second job a réutilisé le libellé de lot deux jours plus tôt, et rien dans le texte de l'instruction ne le dit. Personne n'a été négligent : Sam a décrit la tâche avec exactitude, l'assistant a écrit ce qu'on lui demandait d'écrire, et Jordan a relu ce qui lui était montré. Ce qui manquait, c'était une étape entre « approuvé » et « commit », un endroit où quelque chose regarde ce que l'instruction va faire aux données, et pas seulement ce qu'elle dit.

Les chapitres 3 à 8 ont décidé qui peut détenir un grant, par quel chemin et pour combien de temps. Ce chapitre ajoute la partie Exécution du modèle en cinq parties : l'étape où la gateway prend en charge chaque instruction, au lieu de remettre une session et de faire confiance à quiconque la détient. La question à laquelle il répond est la suivante : qu'est-ce qui doit se tenir entre une instruction approuvée et les données, quand l'approbation seule ne peut pas dire ce que fera l'instruction ?

Le problème, et pourquoi il est difficile

Un session handout donne un login à une personne et laisse son client parler directement à la base de données. Tout ce qui suit est l'affaire du client : ce qui est saisi, combien de lignes cela touche, si cela fait un commit sur-le-champ. Le grant borne les tables et les colonnes accessibles, et rien ne borne ce que fait une instruction donnée à l'intérieur de ce périmètre. Dans l'état initial, un client de bureau soumet tout ce qui est saisi et en fait immédiatement le commit.

L'exécution médiée inverse cet arrangement. L'instruction va à un composant qui n'est pas le client de la personne, et ce composant décide si elle s'exécute, avec quelles limites, et quand elle fait l'objet d'un commit. La gateway du Chapitre 7 est déjà sur le chemin des données ; c'est donc l'endroit naturel pour cela. La personne écrit toujours l'instruction, ou une automatisation le fait ; ce qui change, c'est qu'aucune instruction n'atteint les données sans passer par la boucle de la section suivante.

C'est un continuum, pas un interrupteur. La forme la moins chère est un session handout avec quelques paramètres que la gateway fait respecter, comme un timeout d'instruction et un row cap. La forme suivante vérifie chaque instruction avant de la transmettre. La forme la plus médiée approuve chaque instruction individuellement et l'exécute depuis un worker, si bien que la personne ne détient jamais de session de base de données pour l'écriture. Certains outils approuvent chaque requête plutôt qu'une fenêtre de temps. Au moment où j'écris ces lignes, QueryProxy, par exemple, fait approuver l'instruction par une seconde personne et la fait exécuter par un worker, si bien que le développeur ne détient jamais le credential de la base de données. Cela échange la rapidité d'une session interactive contre une approbation liée à l'instruction exacte, et d'autres outils d'approbation des requêtes fonctionnent de façon similaire.

La médiation est difficile pour trois raisons. La première est que SQL est expressif : le même effet peut s'écrire de nombreuses façons, et savoir si une instruction est « dangereuse » dépend souvent de données que le vérificateur n'a pas lues. La deuxième est que chaque vérification a un coût en latence, en refus injustifiés et en personnes qui apprennent à la contourner. La troisième est qu'un vérificateur auquel on fait trop confiance devient la chose que personne ne relit. La position de ce chapitre est que l'analyse des instructions réduit les erreurs et n'est pas la frontière d'auto-

risation. Les grants sont la frontière. Le rôle de la gateway est de rendre l'instruction erronée courante difficile à exécuter par accident, et de faire porter à chaque écriture une borne que quelque chose d'autre que son auteur fait respecter.

La boucle de la gateway pour chaque instruction

Chaque instruction qu'envoie une session, humaine ou non, passe par la même boucle. L'ordre compte plus que n'importe quelle étape prise isolément ; la voici donc en pseudocode avant que les éléments soient expliqués.

```
function handleStatement(session, statement):
    if statement.changesSetting(guardedSettings):
        return refuse(session, "guarded setting")
    verdict = evaluateStatement(session, statement)
    // the record comes first: a statement that cannot be recorded
does not run
    try:
        recordIntent(session, statement, verdict)
    on RecordUnavailable:
        return refuse(session, "record unavailable")
    if verdict.action == "deny":
        return refuse(session, verdict.reason)
    if verdict.action == "gate":
        return routeToGate(session, statement, verdict)   // approval,
or the product path
    if statement.isRead:
        return streamRows(session, statement, cap: session.rowCap)
// cursor closes at the cap
    // auto-commit is off: the runner owns the transaction of a fix,
the gateway owns the rest
    tx = transactionFor(session)
    result = execute(tx, statement)
    tx.rowsSoFar = tx.rowsSoFar + result.rows      // counted across
every statement of the fix
    if tx.rowsSoFar > verdict.bound:
        failTransaction(tx)                        // its owner can
only roll back
        return refuse(session, "more rows than approved")
    showRowCount(session, tx.rowsSoFar)            // before COMMIT,
while undo is free
    if tx.ownedByGateway:
        if not countAccepted(session, tx.rowsSoFar, verdict.bound):
            rollback(tx)
            return refuse(session, "count not accepted")
        commit(tx)                                 // the commit point
for a person or a migration
    recordOutcome(session, statement, result)      // failure is
queued and alerts; a commit stands
    return result
```

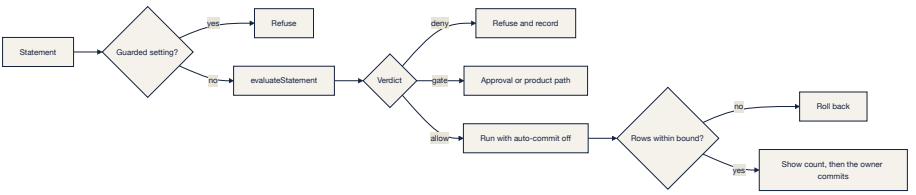
Guarded settings. La gateway ouvre chaque session avec ses limites appliquées : le timeout d'instruction, un mode lecture seule par défaut, le rôle et la limite de connexions selon le type de session (Chapitre 5). Un client peut envoyer un `SET` qui lève n'importe laquelle d'entre elles ; la gateway tient donc une liste de guarded settings et refuse une instruction qui en modifie un. Le qualificatif « non contournable » ne se mérite que là où un mécanisme le fait respecter. Un timeout défini comme valeur par défaut d'un rôle peut être levé par la session, à moins que la gateway ne refuse le `SET` ou que la plateforme ne le fige ; une limite qui n'a ni l'un ni l'autre est

donc une valeur par défaut et non une garantie. Le coût est que certains clients envoient de tels paramètres à la connexion et cessent de fonctionner jusqu'à ce que quelqu'un les inscrive dans la liste.

Le row cap. Pour les lectures, la gateway compte les lignes au fil du streaming, ferme le curseur au cap, en informe la personne et enregistre l'événement. Le cap ne dit pas qu'une personne ne peut voir qu'un certain nombre de lignes par jour, et une personne déterminée peut parcourir une table par tranches. Il dit qu'une seule requête erronée ne fait pas descendre une table entière sur un ordinateur portable. Les extractions volumineuses légitimes ont leur place sur le chemin d'export du Chapitre 13 et sur la copie analytique masquée du Chapitre 12, où le palier décide qui approuve.

Auto-commit désactivé pour les écritures. Une session d'écriture exécute chaque écriture dans une transaction explicite. Pour une personne ou une migration, c'est la gateway qui l'ouvre ; pour une correction de données, c'est le runner du Chapitre 10 qui l'ouvre, et la gateway transmet chaque instruction de la correction dans cette unique transaction. Dans les deux cas, la gateway compte les lignes affectées sur l'ensemble des instructions et les compare à la borne, tandis que la décision de faire le `COMMIT` reste au propriétaire de la transaction : une gateway qui ferait de son côté un commit de chaque instruction d'une correction casserait la transaction unique du runner et ferait de son dry-run une modification réelle. Quand le comptage dépasse la borne, la gateway refuse l'instruction et marque la transaction comme échouée, si bien que le propriétaire ne peut que faire un rollback. C'est la forme bon marché de réversibilité du Chapitre 1 : une mise à jour censée toucher une ligne et qui en annonce quatre cents est arrêtée tant que l'annulation est encore gratuite. Pour une session interactive, « affiché » signifie qu'une personne voit le comptage et le confirme. Pour le runner et pour l'automatisation, « affiché » signifie que la gateway compare le comptage à la borne enregistrée par l'approbation et fait échouer la transaction au moindre excédent. Le coût est l'étape supplémentaire, ainsi que les verrous de la transaction, qui restent détenus pendant que la personne décide ; une demande de confirmation à laquelle personne ne répond détient un verrou de ligne, la demande a donc un timeout qui déclenche un rollback.

L'enregistrement d'abord. La gateway écrit l'intention avant que l'instruction s'exécute et le résultat après. Aucune transaction ne couvre à la fois le log de la gateway et la base de données ; c'est donc l'ordre qui porte la garantie. Si l'intention ne peut pas être écrite, l'instruction ne s'exécute pas. Si le résultat ne peut pas être écrit après un commit, le commit est maintenu, l'entrée est mise en file d'attente et l'équipe plateforme est alertée ; un log qui ne fonctionne que lorsque tout va bien ne mérite pas ce nom. Le Chapitre 14 explique où vit l'enregistrement.



Classer une instruction

La gateway décide quoi faire d'une instruction dans `evaluateStatement`. Elle classe l'instruction selon ce qu'elle fait et à quel endroit, consulte la matrice présentée deux sections plus loin et, pour une écriture que la matrice autorise avec une borne, trouve la borne et lui compare l'estimation du planner. Chaque refus et chaque gate proviennent de la matrice ; la fonction n'a pas de retour anticipé qui lui soit propre en dehors de ceux du parseur.

```
function evaluateStatement(session, statement):
    parsed = parse(statement)
    if parsed.failed or parsed.isMultiStatement:
        return deny("cannot be analyzed")            // fail closed
    // nine classes, one per matrix row; the most specific class wins
    class = classify(parsed)
    cell = matrix[class][session.kind]               // deny, gate,
allow or allow with a bound
    if cell.action != "allow":
        return cell                                  // a deny, or a
gate with its route
    if class == "read":
        return allow(cap: session.rowCap)
    if cell.bound == "fix":
        binding = checkBinding(session, statement)   // the next
statement of the approved fix
        if binding.action == "deny":
            return binding
        bound = binding.bound
    else:
        bound = session.writeBound()   // the break-glass row cap, or
the backfill rule's limit
    if bound is none:
        return deny("no approved count")
    estimate = plannerEstimate(parsed)               // asks the
planner; runs nothing
    if estimate > bound * tolerance:
        return deny("estimate exceeds the approved count")
    return allow(bound: bound)
```

Trois choix méritent d'être défendus. Premièrement, une instruction que le parseur ne sait pas lire est refusée. Si la gateway exécutait ce qu'elle ne peut pas analyser, l'analyse ne s'appliquerait qu'aux instructions qui n'ont jamais posé problème. Un client qui envoie des lots d'instructions dans une seule chaîne ne fonctionne plus sous cette règle, et c'est un coût à nommer dans la migration ; la réponse est de les envoyer une par une.

Deuxièmement, la classification regarde l'effet et non le nom de la personne ou de l'outil. Un `DELETE` dans une migration et un `DELETE` dans une session JIT appartiennent à la même classe, et la matrice décide de ce que chaque lane peut en faire. Un seul classificateur avec une seule table, plutôt qu'un ensemble de règles distinct par chemin, voilà ce qui garde la politique assez petite pour être relue.

Troisièmement, l'estimation du planner est comparée au nombre attendu, mais seulement comme un refus précoce. Le planner estime à partir de statistiques, et les statistiques vieillissent ; une estimation peut être trop

basse aussi facilement que trop haute. L'estimation permet à la gateway de refuser une instruction manifestement fausse avant qu'elle prenne des verrous. Le contrôle, c'est le comptage réel, comparé à l'intérieur de la transaction avant le `COMMIT`. La tolérance est un paramètre que l'équipe plateforme fixe et enregistre, et je commencerais serré pour ne l'assouplir que lorsque les refus injustifiés sont mesurés, pas supposés.

Ce que l'analyse statique ne peut pas voir

Le texte d'une instruction dit peu de chose de son effet sur les lignes. Un parseur peut voir qu'un `WHERE` manque, qu'un prédicat est toujours vrai, qu'une instruction touche une table qu'elle ne devrait pas toucher, ou qu'elle appelle une fonction. Il ne peut pas voir combien de lignes un prédicat sélectionne aujourd'hui, ce qui est le problème de l'ouverture : un filtre valide, sur un libellé qui correspond désormais à plus de lignes que quiconque ne l'attendait.

Plusieurs autres cas échappent à ce que l'analyse peut trancher. Un filtre peut être valide et pourtant large, sélectionnant la majeure partie d'une table à cause des données et non de la syntaxe. Une jointure peut multiplier les lignes. Une vue ou une fonction peut cacher une écriture ou une lecture plus large derrière un nom d'apparence inoffensive, et la gateway ne voit pas l'intérieur du corps d'une fonction que le rôle a le droit d'appeler. Un effet de bord déclenché par un trigger survient après l'instruction et est invisible dans son texte. Et une instruction peut avoir exactement la bonne forme et le mauvais sens : les bonnes lignes selon le filtre, les mauvaises selon l'intention, ce qu'aucun outil ne peut juger.

Des outils open source comme QueryProxy vérifient les instructions à la soumission, en rejetant un `UPDATE` ou un `DELETE` dont le `WHERE` est absent ou toujours vrai et en plafonnant les `SELECT` non bornés ; au moment où j'écris ces lignes, sa propre documentation décrit ce parseur comme une couche de défense plutôt que comme la frontière d'autorisation, ce qui est tout l'objet de cette section, et d'autres gateways et outils d'approbation adoptent des approches similaires.

Il en découle une répartition du travail. L'analyse attrape les erreurs courantes à moindre coût et tôt. Les grants (Chapitre 3) fixent ce qui est accessible tout court, si bien qu'une instruction qui passe entre les mailles

reste bornée par le rôle. La vérification du comptage dans la transaction attrape le cas où les données ont contredit l'auteur. L'approbation, avec une seconde personne qui lit l'instruction et le nombre attendu, attrape le cas où la forme était bonne et l'intention mauvaise, une partie du temps. Aucun de ces contrôles n'est complet, et la conception repose sur leur recouvrement sans prétendre qu'un seul d'entre eux soit la frontière.

La même limite façonne les paliers d'export du Chapitre 13. La gateway calcule le palier d'un export à partir des colonnes sources que lit la requête, et non à partir des noms des colonnes du résultat, et une source qu'elle ne peut pas résoudre compte comme le palier exportable le plus élevé ; là où même cela ne peut pas être décidé, l'export est refusé. C'est la même règle d'échec en mode fermé que celle du parseur : ce que l'analyse ne peut pas lire, elle ne le laisse pas passer.

La matrice deny, gate et allow

Chaque classe d'instruction reçoit l'une de trois réponses dans chaque lane. **Deny** signifie que la gateway refuse et enregistre le refus. **Gate** signifie que l'instruction ne s'exécute pas telle qu'elle a été envoyée : elle a besoin d'une approbation qu'elle n'a pas encore, ou elle appartient à une autre route. **Allow** signifie qu'elle s'exécute sous les limites de la boucle ci-dessus ; là où une cellule dit « borne », l'écriture n'est autorisée qu'à l'intérieur d'une borne issue de la source que nomme la cellule, et le comptage réel lui est comparé avant le `COMMIT`. Les colonnes sont les lanes que le livre a décrites, et les lignes sont les classes d'instructions que produit le classificateur.

Classe d'instruction	Session de lecture humaine	Écriture JIT	Break-glass	Runner de correction de données	Migration
Lecture, dans le row cap	Allow	Allow	Allow	Allow	Allow
Écriture bornée, dans le nombre approuvé	Deny	Allow, borne liée à la correction (runner uniquement)	Allow, borne liée à un row cap (note obligatoire)	Allow, borne liée au hash et au nombre approuvés	Allow, borne liée à la règle de backfill

Classe d'instruction	Session de lecture humaine	Écriture JIT	Break-glass	Runner de correction de données	Migration
Écriture non bornée	Deny	Deny	Deny	Deny	Gate : approbateur nommé
Changement de lien de propriétaire	Deny	Gate : chemin produit	Gate : chemin produit	Gate : chemin produit	Gate : approbateur nommé
Écriture dans une colonne Restricted	Deny	Gate : chemin produit	Gate : chemin produit	Gate : chemin produit	Gate : approbateur nommé
Export	Gate : selon le palier	Deny	Deny	Deny	Gate : approbateur nommé
Changement de schéma (DDL)	Deny	Deny	Deny	Deny	Allow : lane revue
Changement de privilèges	Deny	Deny	Deny	Deny	Gate : revue plateforme
Écriture dans `ledger_entries` ou `audit_events`	Deny	Deny	Deny	Deny	Gate : revue plateforme

La lire ligne par ligne montre la conception. La session de lecture humaine n'a pas de chemin d'écriture, si bien que chaque classe d'écriture y est refusée, et le seul gate est le palier d'export. Une session d'écriture JIT ne transmet les écritures que depuis le runner, si bien que sa cellule d'écriture bornée est un allow assorti d'une liaison et non une porte ouverte. Le break-glass est la seule lane où une personne écrit directement, c'est pourquoi son écriture bornée n'est autorisée que sous un row cap et avec une note, et c'est pourquoi la taille de ce cap est la décision de dimensionnement la plus difficile du livre : une urgence n'est pas la tâche planifiée, donc aucun cap ne peut être déduit de la tâche. Le cap doit être fixé

d'après ce que le rôle break-glass aurait plausiblement besoin de réparer, et revu après chaque usage. Une écriture non bornée y est refusée. Ce qui reste, c'est une écriture qui tient dans le cap et touche les mauvaises lignes, un risque accepté que la revue existe pour exposer, comme l'énonce le Chapitre 8 ; le Chapitre 8 dit aussi ce qui vaut aujourd'hui, avant qu'une gateway impose le cap.

Le changement de lien de propriétaire mérite sa propre ligne. Un `UPDATE` de `customer_id` sur `bank_accounts` ou `payment_tokens` passe le plafond JIT, parce que la colonne relie une ligne à son propriétaire et n'est pas Restricted. Déplacer la ligne vers un autre client change malgré tout à qui appartient le compte de versement ou le token de paiement. Le classificateur en fait donc une classe à part entière, et toutes les lanes sauf la migration répondent par un gate qui l'oriente vers le chemin produit : `admin-console` sollicite `payments-service`, une seconde personne approuve, et la modification est enregistrée dans `audit_events`. Une migration dans la lane normale la soumet à un gate pour un approbateur nommé, comme le fait le Chapitre 11 pour toute instruction signalée, et la lane accélérée la refuse ; un changement de schéma n'a guère de raison de réattribuer la propriété, et si une réattribution en masse s'avère un jour nécessaire, c'est une fonctionnalité produit à construire et non une instruction à exécuter. Les écritures dans les colonnes Restricted (`iban`, `account_holder_name` et `token_value`) suivent la même route, c'est pourquoi aucune lane ne permet à un humain de les écrire en SQL brut. Le seul cas de migration est le déplacement des colonnes Restricted vers les tables de secrets séparées du Chapitre 11, qui va à un approbateur nommé et non à un refus.

La colonne migration correspond à ce que décrit le Chapitre 11. Son classificateur signale des instructions pour un approbateur nommé ou pour une revue plateforme, et ne refuse que dans la lane accélérée. Dans ce tableau, les signalements apparaissent comme des gates là où une personne doit dire oui, et les allows sont le changement de schéma revu et le backfill qui reste dans sa règle. Les tables append-only suivent le même principe : une migration qui les touche va à l'équipe plateforme, et les autres lanes la refusent. Un second ensemble de règles pour les instructions interdites serait une seconde politique à maintenir en phase avec celle-ci ; le pipeline de migration réutilise donc le classificateur et la matrice et n'ajoute que ses règles de lane.

La matrice est volontairement petite. Neuf classes et cinq lanes tiennent sur une page, et une politique qui tient sur une page est lue. Chaque nouvelle classe ou lane qu'on y ajoute est une décision que quelqu'un doit défendre, et la défaillance habituelle est une matrice qui se dote d'une exception à chaque incident jusqu'à ce que personne ne puisse dire ce qu'elle interdit. Je la garderais sous contrôle de version, relue comme du code, avec l'équipe plateforme comme propriétaire et l'équipe data comme relecteur obligatoire pour les lignes d'écriture.

Lier l'approbation à l'instruction

Une approbation qui couvre « une écriture sur `invoices` » laisse passer n'importe quelle écriture sur `invoices`. L'approbation liée à l'instruction signifie que l'approbation nomme cette instruction, et que la gateway refuse toute autre. Le Chapitre 8 décrit la demande et le Chapitre 10 le runner ; cette section énonce, une seule fois, la règle qui les relie.

Une demande d'écriture porte un `fix_id` et un `approved_hash`. Le hash est celui des instructions qui ont été relues, calculé sur l'ensemble quand une correction en compte plusieurs, et sa source compte davantage que la vérification. Il provient de l'enregistrement de merge de la correction, ou de l'enregistrement de la demande approuvée, et jamais de l'entrée du runner. Si le runner fournissait son propre hash, il vérifierait son propre travail, et une modification de l'instruction arriverait accompagnée d'un hash correspondant. Un hash qui trouve son origine dans l'enregistrement qu'une seconde personne a approuvé est un hash dont quelqu'un d'autre que l'auteur s'est porté garant.

```
function checkBinding(session, statement):
    request = loadApprovedRequest(session.requestId)     // from the
store, not from the caller
    if request is none or request.status != "approved":
        return deny("no approved request")
    if request.fixId is none:
        return deny("no fix named")                      // a SELECT-
only request has no fix and no hash
    fix = loadMergeRecord(request.fixId)                 // the
statements, read from the record
    if hashOf(fix.statements) != request.approvedHash:
        return deny("record differs from the approved hash")
    next = fix.statements[session.transaction.statementCount]
    if normalize(statement) != normalize(next):
        return deny("statement differs from the approved one")
    return allow(bound: fix.expectedRows)               // the count
comes from the hashed record
```

La gateway vérifie une instruction à la fois, mais le hash couvre l'ensemble : elle charge les instructions depuis l'enregistrement de merge, confirme que leur hash est celui qui a été approuvé, et n'accepte que l'instruction suivante dans cet ordre. La borne est le nombre de lignes attendu figurant dans le même enregistrement, et non un nombre que porte la demande ; le hash protège donc aussi le comptage. Le commit reste dans la transaction du runner, si bien qu'une correction de plusieurs instructions s'applique entièrement ou pas du tout.

Deux règles complètent la liaison. Dans une session d'écriture JIT, la gateway ne transmet les écritures que depuis la connexion du runner. Le client de la personne peut toujours lire à l'intérieur des triplets accordés, mais une écriture envoyée depuis ce client est refusée, sinon le hash serait une vérification qu'un client pourrait contourner. Un guard n'est un mécanisme que si la session ne peut pas écrire d'une autre façon. La seconde règle est le cas inverse : une demande JIT qui ne contient que des triplets `SELECT` n'a ni correction ni hash, il n'y a donc rien à quoi une écriture puisse être liée, et l'écriture est refusée. Une demande de lecture ne peut pas devenir une demande d'écriture par l'envoi d'une autre instruction.

Là où la gateway se contente de fournir un credential et où le client se connecte directement, rien de tout cela n'est applicable, et le livre le dit sans détour : le hash enregistre alors une intention et n'empêche rien. La liaison est une propriété du chemin des données, et elle n'existe que là où la gateway s'y trouve.

Le cas du runbook est celui qui comporte un modèle. L'enregistrement de merge d'un runbook stocke le hash du modèle d'instruction, et la demande ne porte que la liste des clés. Le runner substitue les clés dans le modèle stocké. Dans ce cas, la gateway vérifie le hash par rapport au modèle et non par rapport au texte substitué : elle confirme que l'instruction soumise est le modèle approuvé, avec uniquement la liste des clés renseignée, et elle utilise le nombre de clés comme borne, avec les triplets de la demande. La gateway ne prend jamais une instruction ou un hash de l'appelant qui veut que le travail soit fait.

Les appelants qui ne sont pas des personnes

Les jobs, les scripts de runbook, les bots ChatOps et les agents d'IA soumettent tous des instructions, et aucun d'eux n'est une personne. La règle de ce chapitre est qu'ils passent par la même boucle, en tant que principals à part entière. Il n'existe pas de chemin allégé pour l'automatisation, parce que l'automatisation est l'endroit où une erreur se répète à la vitesse d'une machine.

Un appelant automatisé a une workload identity fournie par la plateforme, et non la session d'une personne, ainsi qu'un mode lecture seule par défaut. Il ouvre sa session sous son propre principal, si bien que ses lectures sont soumises au même row cap, au même timeout et aux mêmes guarded settings que celles de n'importe qui. S'il veut écrire, il suit la même route qu'une personne : il nomme une correction, un approbateur qui est une personne autre que celle qui a déclenché l'automatisation l'approuve, et ses écritures passent par le runner avec un nombre attendu. La règle des deux personnes du Chapitre 8 vaut ici aussi : quiconque a demandé à l'automatisation de rédiger la correction ne peut pas être celui qui l'approuve, et il n'y a pas d'approbation automatique par un appelant qui n'est pas une personne. Un agent qui propose une instruction la propose, et l'approbation, c'est un humain qui la lit.

L'enregistrement nomme les deux parties : l'automatisation qui a soumis l'instruction et la personne qui l'a déclenchée. C'est la moitié qui relève de l'imputabilité. Un enregistrement qui dirait seulement « c'est l'assistant » arrêterait la piste à une machine, et un enregistrement qui dirait seule-

ment « Sam l'a demandé » cacherait quel outil a transformé la demande en SQL. Les deux noms sont stockés, et la revue des accès interroge sur les deux.

Un agent d'IA ajoute une préoccupation que les autres appelants n'ont pas : ses instructions sont générées à partir de texte, dont il n'a pas écrit une partie. Un ticket, une ligne de log ou un message de client peut se retrouver dans le contexte qui produit l'instruction. La défense ne consiste pas à lire le texte plus attentivement. Elle consiste en ce que les privilèges de l'agent ne changent pas avec ce qu'il lit. Ses grants sont les mêmes, ses écritures exigent toujours l'approbation par une personne de l'instruction exacte, et une instruction qui arrive par une route que l'agent ne devrait pas avoir, comme une écriture hors du runner, est refusée par la même matrice. Quoi qu'on ait persuadé l'agent de proposer, il ne peut pas l'approuver, et il ne peut pas l'exécuter sans la borne.

Pour un agent en lecture seule, les limites utiles sont les limites ordinaires : un row cap serré, un timeout court, une vue ou une copie masquée à la place des colonnes Restricted, et un budget par principal, de sorte qu'une boucle qui déraille épuise sa propre allocation et non l'allocation commune. Pour un agent qui écrit, la limite utile est que ses écritures sont peu nombreuses et petites, parce que chacune a besoin d'une personne. Si cela rend l'agent trop lent pour être utile en écriture, la bonne conclusion est que la tâche doit devenir une fonctionnalité produit que l'agent appelle, et non que l'approbation doit être assouplie.

Compromis et coûts

La médiation coûte du temps. Chaque instruction passe par un composant de plus, et une écriture soumise à un gate attend une personne. Pour les lectures interactives sur une réplique, la latence ajoutée est faible et largement invisible. Pour les écritures, c'est tout l'intérêt : le délai est l'endroit où a lieu le second regard. Je ne le supprimerais pas pour les cas urgents ; je rendrais le gate de la lane break-glass rapide et bruyant, et je le revisiterais après coup.

Elle coûte de la compatibilité. Une gateway qui analyse les instructions et refuse ce qu'elle ne sait pas analyser cassera certains clients, certains drivers et certaines habitudes. Refuser les instructions `SET`, les chaînes multi-

instructions et la syntaxe inhabituelle produit une liste de choses qui cessent de fonctionner. Cette liste est plus courte si elle est établie avant le rollout, à partir des enregistrements d'instructions des sessions existantes, que si elle est découverte au fil des tickets de support. Prévoyez une période où la gateway enregistre ses verdicts mais ne refuse que les cas les plus clairs, et passez à l'application classe par classe à mesure que les refus injustifiés diminuent.

Elle coûte un composant qui doit être disponible. La vérification des instructions se trouve sur le chemin de chaque écriture, et le Chapitre 7 explique pourquoi ce composant doit rester petit. Là où la vérification est en panne, le comportement par défaut qui borne les dégâts est que les écritures s'arrêtent et que les lectures sur la réplique continuent. Une vérification qui échoue en mode ouvert apprend à l'équipe que la vérification est facultative, et la première panne met fin à la politique.

Elle coûte une politique à maintenir. La matrice, la liste des guarded settings, la tolérance et les row caps sont tous des paramètres dont quelqu'un est responsable. Une politique sans responsable dérive vers ce dont le dernier incident a eu besoin. La matrice est l'unique artefact à garder petit, relu et versionné.

Il existe aussi un coût que l'ingénierie ne peut pas payer : l'analyse donne un faux sentiment de sécurité. Une équipe qui dispose d'un parseur rejetant les `UPDATE` sans `WHERE` peut se mettre à considérer une instruction qui est passée comme une instruction qui ne peut pas faire de mal. La matrice dit ce que la gateway refuse, et l'ouverture de ce chapitre montre ce qu'une instruction qui passe peut encore faire. La vérification du comptage et la seconde personne existent pour ce cas, et il ne faut pas les alléger sous prétexte que le parseur est bon.

Erreurs courantes

Traiter l'analyseur comme la frontière. L'analyseur est une couche qui attrape l'instruction erronée courante. Les grants décident de ce qui est accessible, et une conception qui fait porter ce poids à l'analyseur échoue dès la première instruction valide et fausse.

Laisser le runner ou l'appelant fournir le hash. Un hash qui arrive avec l'instruction prouve seulement que l'instruction correspond à elle-même. Le hash provient de l'enregistrement de merge ou de la demande approuvée, et la vérification s'exécute dans la gateway, où l'appelant ne peut pas la modifier.

Laisser l'auto-commit activé pour les sessions d'écriture. Avec l'auto-commit activé, le nombre de lignes est un compte rendu après coup. Le désactiver, et afficher le comptage avant le `COMMIT`, c'est ce qui rend un mauvais comptage bon marché.

Échouer en mode ouvert quand le parseur ne sait pas lire une instruction. Exécuter ce qui n'a pas pu être analysé fait que l'analyse ne couvre que les cas inoffensifs. Refusez-le, et tenez une courte liste des clients à modifier.

Un chemin allégé pour l'automatisation. Donner aux jobs et aux agents un raccourci qui contourne la boucle, parce qu'on leur fait confiance ou qu'ils sont pressés, retire les contrôles là où le volume est le plus élevé. L'automatisation passe par la même boucle sous son propre principal.

Une matrice qui se dote d'une exception à chaque incident. Chaque exception est une décision sans responsable. Limitez le tableau aux classes et aux lanes qui existent, et quand un cas n'y entre pas, changez la conception ou construisez la fonctionnalité.

Prendre un row cap pour une limite d'exposition. Un cap borne une requête, pas ce qu'une personne peut lire en une journée. Si l'exposition est la préoccupation, elle relève du travail de masquage et de paliers des chapitres 12 et 13.

Points clés

- L'exécution, c'est la façon dont une instruction atteint les données : qui la soumet, si quelque chose l'inspecte d'abord, ce qui borne son effet et comment elle fait l'objet d'un commit. Les grants ne répondent pas à ces questions.
- L'exécution médiée place un composant entre l'instruction et les données. La forme la moins chère se résume à quelques paramètres impo-

sés ; la plus forte approuve chaque instruction et l'exécute depuis un worker.

- Chaque instruction, qu'elle vienne d'une personne ou d'une automatisation, passe par une seule boucle : guarded settings, évaluation, enregistrement d'abord, row cap pour les lectures, et auto-commit désactivé pour les écritures, avec le nombre de lignes affiché avant le `COMMIT`.
- L'analyse réduit les erreurs et n'est pas la frontière d'autorisation. Les grants sont la frontière, et le comptage réel des lignes vérifié dans la transaction attrape ce que le parseur et le planner ne voient pas.
- Une seule petite matrice croisant classes d'instructions et lanes donne chaque réponse sous la forme deny, gate ou allow. Le changement de lien de propriétaire et les écritures dans les colonnes Restricted sont soumis à un gate et orientés vers le chemin produit.
- L'approbation est liée à l'instruction : l'`approved_hash` provient de l'enregistrement de merge ou de la demande approuvée, jamais du runner, et dans une session d'écriture JIT la gateway ne transmet les écritures que depuis le runner.
- Les jobs, les runbooks, ChatOps et les agents d'IA sont des principals dotés de leur propre identité, d'un mode lecture seule par défaut et de la même boucle. Leurs enregistrements nomment à la fois l'automatisation et la personne qui l'a déclenchée.

La boucle et la matrice disent ce que la gateway fait d'une instruction. Elles ne disent pas comment écrire une correction qui mérite d'être approuvée : comment prouver le comptage, conserver un snapshot, planifier le rollback et la faire relire par une seconde personne. Le Chapitre 10 aborde ce travail.

Chapitre 10 — Requêtes ad hoc et corrections de données

Imaginez un jeudi après-midi chez Ledgerly. Un bug dans `invoice-worker` a été corrigé et déployé, mais pendant qu'il était en production il a laissé un lot de factures dans le status `sending`, alors que les e-mails étaient partis. Le support pose des questions à leur sujet, les relances des clients sont suspendues, et Robin, d'astreinte côté backend, voit la forme de la réparation en une ligne : remettre le status à `sent` pour les factures concernées, et rien d'autre.

Robin ouvre une session via l'access gateway. La session par défaut est en lecture seule, donc la ligne ne peut pas s'exécuter. Robin doit demander un accès en écriture, et cette demande doit nommer des tables et des colonnes. Quelqu'un d'autre doit regarder l'instruction avant qu'elle ne touche le primary. Pendant ce temps, l'horloge tourne, et la réparation est assez petite pour que le processus qui l'entoure semble plus gros que le travail lui-même.

Ce chapitre ajoute une étape de l'état cible : les écritures passent par une procédure de correction de données relue et par des grants just-in-time dans le plafond de `human_jit_write`. Le Chapitre 7 a donné aux humains une seule porte, et le Chapitre 8 a construit les grants qui l'ouvrent pour une durée limitée. Ici, nous nous demandons ce qu'une personne doit faire une fois la porte ouverte : comment lire, comment écrire, et comment rendre le second type de travail assez sûr pour qu'il puisse se faire calmement un jeudi après-midi.

Dans le modèle en cinq parties du Chapitre 1, ce chapitre porte sur l'**Exécution**. Robin détient déjà tous les credentials dont la réparation a besoin, donc le risque n'est pas dans la question de savoir qui détient un credential. Il est dans ce que le grant autorise, dans ce qu'une instruction fera quand elle s'exécutera, dans qui l'a lue d'abord, et dans les preuves qui restent. La question du chapitre est la suivante : quelle est la manière la plus

petite et la plus facile à relire pour une personne de modifier des données de production, et quand le changement devrait-il cesser complètement d'être le travail d'une personne ?

Le problème, et pourquoi il est difficile

Tout système qui contient des données réelles finit par atteindre un état que son propre code ne peut pas réparer. Un worker a planté entre deux étapes. Une migration a laissé une colonne à moitié remplie. Un client a été fusionné avec un autre par erreur. L'application n'a pas d'endpoint pour « remettre cela en ordre », parce que personne ne s'attendait à en avoir besoin, et écrire l'endpoint prendrait plus de temps que la réparation. Alors une personne écrit une instruction et l'exécute sur la production. C'est une **correction de données** : une modification des données de production faite par une personne en dehors du chemin normal de l'application, pour réparer un état que l'application ne peut pas réparer.

La première difficulté est que l'outil dangereux et l'outil utile sont le même outil. Une session SQL qui peut répondre à « combien de factures sont bloquées ? » peut, avec un mot-clé de plus, les modifier toutes. Dans l'état initial du Chapitre 1, le client graphique détient le login admin partagé, fait un commit automatique de chaque instruction et exécute le texte qui est sélectionné, quel qu'il soit. La distance entre une question et un changement irréversible est d'un mot.

La deuxième difficulté est que la personne qui fait le travail est généralement sous pression et a généralement raison. Un processus qui traite chaque correction comme une menace apprend aux gens à le contourner, et le contournement est pire que le risque initial : un grant permanent, un mot de passe partagé, une copie de la production sur un ordinateur portable. Un processus qui traite chacune comme de la routine n'apprend à personne à regarder deux fois, et l'instruction sur cent qui est fausse atteint toutes les lignes. Le chemin sûr doit être le chemin facile.

La troisième difficulté est que l'unité de risque n'est pas la personne mais l'instruction, et qu'une instruction ne porte presque aucune information sur sa propre taille. `UPDATE invoices SET status = 'sent' WHERE id = 'inv-0042'` et la même ligne dont la clause `WHERE` a été coupée ne dif-

fèrent que de quelques caractères. Rien dans le texte ne dit « ceci va toucher neuf mille lignes » ; seules les données le disent, et on ne les lit qu'après coup, à moins que le processus ne pose la question d'abord.

La quatrième est que la réponse ne peut pas être seulement une politique. « Comptez les lignes avant de faire l'update » se lit une fois et se suit jusqu'au premier jour chargé ; il faut donc l'intégrer au chemin : un runner qui refuse de faire un commit quand le décompte est faux, et une session qui ne peut écrire d'aucune autre manière.

Les requêtes ad hoc ont un problème plus discret. Une **requête ad hoc** est une requête qu'une personne écrit et exécute une seule fois, en dehors de toute application. Dans une session en lecture seule, elle ne peut rien modifier, mais elle peut lire beaucoup, elle peut coûter cher à la base de données, et ses résultats quittent le système dans une capture d'écran, un tableur ou un message de chat.

Le défaut: une session en lecture seule

Une session humaine commence comme un login membre de `human_readonly`, connecté via l'access gateway à `ledgerly_replica_2`. Elle peut lire les tables que couvre le rôle, avec les colonnes Restricted retenues et les colonnes Confidential masquées là où le Chapitre 12 le prévoit. Elle ne peut pas écrire, et rien de ce que tape la personne n'y changera quoi que ce soit.

La plupart des investigations tiennent dans cette session, et la personne n'a besoin de rien demander ni d'attendre personne. Je défendrais ce défaut sur le seul critère du coût : une conception dans laquelle la tâche courante ne demande aucune approbation est une conception dans laquelle l'approbation de la tâche peu courante garde un sens.

Trois habitudes protègent la réplique partagée et ses résultats. **Bornez la requête** : le timeout d'instruction au niveau du rôle (Chapitre 5) et le row cap de la gateway (Chapitre 7) arrêtent les cas incontrôlés, et sélectionner d'abord des décomptes et des échantillons garde les exports accidentels petits. **Sélectionnez le minimum qui répond à la question** : `id, status, due_on` vaut mieux que `SELECT *`, en partie parce que c'est plus rapide et en partie parce que cela garde les colonnes Confidential hors du résultat, sauf si la question en a besoin. **Traitez les résultats comme des**

copies de données : un résultat collé dans un ticket ou un tableur est une copie de données de production soumise à d'autres règles d'accès (le Chapitre 13 traite ces copies comme des portes latérales) ; collez donc des identifiants et des décomptes, pas des lignes au contenu personnel, et nommez la requête qui les a produits.

Une réserve compte pour la section suivante. La réplique a un certain retard sur le primary, généralement faible et parfois non. Un décompte pris sur elle est une estimation de ce que contient le primary à cet instant. Pour une investigation, cela suffit. Pour décider combien de lignes une écriture touchera, non, et la procédure de correction de données ci-dessous prend le décompte faisant foi sur le primary, dans la transaction qui fait l'écriture.

Écrire est un autre acte

Une lecture peut être répétée, abandonnée et ignorée. Une écriture change ce que voit tout autre principal, y compris les services qui tournent à ce moment et qui agiront sur le nouvel état avant que quiconque le vérifie. Un `status` mis à jour à tort peut déclencher un e-mail de relance ; un `due_on` mis à jour à tort peut changer ce qu'on annonce à un client comme montant dû. C'est pourquoi le livre donne à l'écriture son propre chemin, avec sa propre approbation, sa propre limite de temps et son propre enregistrement.

Ce chemin est l'accès just-in-time du Chapitre 8, avec une propriété qui compte ici : une demande d'écriture est liée à une correction relue. La demande porte le `fix_id` de la correction et un `approved_hash`, et ses triplets (privilège, table, colonnes) sont dérivés de la correction, et non tapés par le demandeur. Le maximum qu'une demande puisse solliciter est le plafond défini par `human_jit_write`. Trois conséquences déterminent la manière d'écrire la correction.

Premièrement, **la demande est construite à partir de la correction**. Une correction qui met à jour `invoices.status` a besoin d'exactement un triplet : `UPDATE` sur `invoices` pour la colonne `status`. Le guard de la section suivante lit avant d'écrire, et cette lecture utilise le simple `SELECT` sur `invoices` que `human_readonly` détient déjà ; aucun triplet `SELECT` n'est donc nécessaire. L'exception est une correction qui prend un snapshot

d'une colonne Confidential, comme `billing_address` sur `customers` : sous `human_readonly`, la colonne est masquée, et un snapshot de valeurs masquées est inutile pour un rollback ; la demande nomme donc un triplet `SELECT` pour elle, et l'approbateur le voit.

Deuxièmement, **une colonne Restricted n'apparaît jamais dans la demande**. `iban`, `account_holder_name` et `token_value` ne peuvent pas être nommées, et `INSERT` et `DELETE` sur `bank_accounts` et `payment_tokens` ne sont pas disponibles, parce qu'insérer ou supprimer une ligne définit ou détruit des valeurs Restricted. Un autre changement se trouve dans le plafond sans être pour autant une correction de données ordinaire : l'`UPDATE` de `customer_id` sur `bank_accounts` ou `payment_tokens`. La colonne n'est pas Restricted, mais déplacer une ligne change à qui appartient le compte de versement ou le token de paiement. La politique d'exécution soumet ce changement de lien de propriétaire au gate et l'oriente vers le chemin produit décrit plus bas (Chapitre 9). Si une correction a besoin de l'un de ces éléments, ce n'est pas une correction de données.

Troisièmement, **l'approbateur voit l'instruction, pas seulement les triplets**. Le Chapitre 9 énonce cette règle de l'approbation liée à l'instruction et la manière dont la gateway l'applique. Pour une correction de données, l'écran d'approbation montre l'instruction, le filtre et le décompte attendu à côté des triplets, et une demande portant sur toutes les colonnes est une demande d'un autre type de confiance. Les corrections écrites avec une empreinte étroite sont approuvées plus vite, ce qui est l'incitation souhaitée.

L'enregistrement de la demande pour la correction des factures bloquées, dans la notation indépendante du moteur du Chapitre 3 :

```
// request record for the stuck-invoice fix; values are illustrative
access_request:
    id              "req-0187"
    kind            "jit"
    requester       "robin"
    approver        "alex"
    fix_id          "FIX-0031"
    approved_hash   "a41c…07"   // set at approval, from the fix's
merge record
    reason          "reset invoices left in 'sending' by worker bug"
    expires_at      now() + 1 hour

access_request_privilege:      // derived from FIX-0031, not typed
    ("req-0187", "UPDATE", "invoices", "status")
```

Sur les moteurs sans grants au niveau des colonnes, les colonnes Restricted vivent déjà dans des tables séparées, hors du plafond (Chapitre 3). Une demande JIT sur la table principale est alors un grant au niveau de la table, et l'approbateur en a approuvé toutes les colonnes non Restricted, Confidential comprises. Le grant ne limite plus l'empreinte de l'instruction, si bien que l'instruction elle-même porte une plus grande part de la revue.

Anatomie d'une correction de données

Une correction de données est une courte procédure dont l'ordre est fixe. L'ordre compte plus que n'importe quelle étape isolée, parce que chaque étape existe pour rendre la suivante moins coûteuse à réussir.

1. **Énoncez l'intention.** Une phrase : quel état est faux, quel état est juste, et quelles lignes sont dans le périmètre. Si elle ne peut pas s'écrire en une phrase, la correction est probablement deux corrections.
2. **Comptez les lignes concernées avec un `SELECT`.** « Quarante factures » et « neuf mille » sont des conversations différentes, et le décompte est le premier endroit où une erreur se révèle.
3. **Notez le nombre de lignes attendu.** La correction déclare le nombre de lignes qu'elle s'attend à modifier. C'est le guard.
4. **Prenez un snapshot avant.** Les clés primaires et les anciennes valeurs d'exactement les colonnes que la correction va modifier, enregistrées dans un endroit qui survit à la session.

5. **Exécutez le changement dans une transaction.** Comparez le nombre de lignes modifiées avec le nombre attendu. S'ils diffèrent, faites un rollback.
6. **Vérifiez une postcondition dans la même transaction.** Une requête qui prouve que l'état est maintenant celui que décrivait l'intention, par exemple « aucune facture de la liste n'a encore le status `sending` ».
7. **Faites le commit, puis enregistrez.** Le commit est le point après lequel les autres principals voient le changement. L'état après, lu dans la transaction, va dans l'enregistrement à côté du snapshot avant.

Le pseudocode place le guard dans le runner et non dans la mémoire de la personne. Il traite la base de données, le stockage des snapshots et le log d'audit comme des systèmes séparés, si bien qu'aucune transaction ne couvre davantage que la base de données. Le runner reçoit un identifiant de demande et rien d'autre ; la première fonction décide si cette demande peut seulement s'exécuter.

```
function loadApprovedFix(request_id, session):
    request = loadRequest(request_id)       // the gateway's record,
not the runner's input
    if request.status != "approved" or request.expires_at <= now():
        raise RequestNotLive
    if request.requester_login != session.login:    // granted to this
session's login only
        raise RequestNotForThisSession
    fix = loadFixScript(request.fix_id)
    // approved_hash was set at approval from the fix's merge record,
so
    // the script is compared with the record, not with itself
    if hashOf(fix.statements) != request.approved_hash:
        raise ScriptNotReviewed
    if request.privileges != fix.privileges:
        raise GrantsDoNotMatchFix
    return fix
```

La seconde fonction l'exécute, et tout ce qu'elle fait est borné par la première :

```
function runDataFix(request_id, session):
    fix = loadApprovedFix(request_id, session)
    // the authoritative count, on the primary, before anything is
written
    before = readRows(fix.target_filter, fix.columns, using: session)
    if count(before) != fix.expected_rows:
        raise RowCountMismatch
    saveSnapshot(fix.id, before)
    transaction:
        // re-read under the lock: a changed value must fail, not pass
        locked = readRowsLocked(fix.target_filter, fix.columns, using:
session)
        if locked != before:
            rollback()
            raise RowsChangedSinceSnapshot
        // one source: the change runs on the same filter that was
counted
        changed = execute(fix.change, where: fix.target_filter, using:
session)
        if changed != fix.expected_rows or not
fix.postcondition(session):
            rollback()
            raise FixGuardFailed
        after = readRows(keysOf(before), fix.columns, using: session)
        // commit point: other sessions see the change only after this
line
        commit()

    try:
        saveAfterState(fix.id, after)
        auditLogAppend({ actor: session.login, action: "data-fix",
                         request: request_id, fix: fix.id, rows:
changed })
    on AuditLogUnavailable or SnapshotStoreUnavailable:
        queueForRetry(fix.id)
        notify("platform", "fix committed, record pending: " + fix.id)
```

Plusieurs détails de ces blocs sont délibérés.

Le runner ne fournit jamais son propre hash. Il charge la correction que nomme la demande et compare le hash de ses instructions avec l'`approved_hash` de l'enregistrement de la demande, que le Chapitre 9 fait remonter à l'enregistrement du merge. Retaper l'instruction change le texte entre la revue et l'exécution, et la revue s'est attachée au texte antérieur.

Le snapshot est enregistré avant la transaction. Une défaillance du stockage des snapshots arrête alors la correction sans que rien n'ait changé. Le coût est que les lignes peuvent changer entre le snapshot et l'écriture :

un worker peut faire passer une facture de `sending` à `failed` dans cet intervalle, et la clé correspondrait encore au filtre.

La transaction relit les lignes sous un verrou. Elle compare le résultat entier, avec les colonnes que la correction modifie, au snapshot. Toute différence dans l'ensemble des lignes ou dans une valeur interrompt l'exécution avec `RowsChangedSinceSnapshot`, et l'auteur recommence à partir d'un nouveau décompte. Sur les moteurs qui ne peuvent pas verrouiller des lignes par un filtre, la relecture s'exécute quand même dans la transaction ; elle réduit l'intervalle sans le supprimer, et ce qui reste, ce sont le décompte attendu et la postcondition.

L'enregistrement est écrit après le commit. Le stockage des snapshots et le log d'audit sont d'autres systèmes, donc l'état après est enregistré une fois le commit effectué. C'est l'ordre inverse de celui du grant JIT du Chapitre 8, où un grant ne doit pas exister sans son entrée de log.

L'ordre se défend ici parce que la gateway de Ledgerly se trouve sur le chemin des données (voir plus bas) : elle transmet les instructions du runner et enregistre la session, si bien qu'un changement ayant fait l'objet d'un commit est enregistré même si l'append d'audit échoue. On fait un retry de l'entrée et l'équipe plateforme est prévenue. Sur une gateway qui ne fait que distribuer des credentials, il n'y aurait pas de log de session des instructions, et l'entrée d'audit devrait passer en premier.

Un dry-run montre ce qu'un décompte ne peut pas montrer. Le décompte est un `SELECT`, il manque donc ce que l'instruction met en mouvement : un trigger qui écrit ailleurs, une cascade, une règle. Un vrai dry-run exécute le changement dans une transaction puis fait un rollback :

```
function dryRunDataFix(request_id, session):
    fix = loadApprovedFix(request_id, session)
    if count(readRows(fix.target_filter, fix.columns, using: session))
!= fix.expected_rows:
        raise RowCountMismatch      // a wrong filter never reaches
the locks
    transaction:
        countsBefore = countsIn(fix.watch_tables, using: session)
        changed = execute(fix.change, where: fix.target_filter, using:
session)
        report = { rows: changed,
                   sample: sampleAfterValues(fix, using: session),
                   related: diff(countsBefore,
countsIn(fix.watch_tables, using: session)) }
        rollback()          // nothing is kept; the row locks are held
until here
    return report
```

Le rapport contient le nombre de lignes, un échantillon des valeurs après et la variation des décomptes de lignes dans les `watch_tables` de l'en-tête de la correction. Il a besoin d'une demande approuvée et des grants d'écriture ; il ne peut donc pas exister tant que la pull request est en revue. Il s'exécute dans la fenêtre approuvée, détient les mêmes verrous de ligne pendant son exécution, et ne peut pas voir les effets hors de la base de données, comme un e-mail. Là où le coût d'une surprise est élevé, l'auteur l'exécute dans les premières minutes de la fenêtre, joint le rapport à la demande, et l'approbateur le lit avant l'étape du commit, au prix d'une pause.

Un guard n'est un mécanisme que si la session ne peut écrire d'aucune autre manière. La gateway de Ledgerly se trouve sur le chemin des données (Chapitre 7) et, dans une session d'écriture JIT, ne transmet que les écritures venant de la connexion du runner ; la vérification du hash, le décompte et le snapshot ne peuvent donc pas être sautés. La transaction reste celle du runner : la gateway y transmet chaque instruction et compte les lignes sur l'ensemble de la correction, mais ne fait jamais de commit à la place du runner, et c'est pourquoi une postcondition en échec entraîne le rollback de toutes les instructions. Le Chapitre 9 explique comment la gateway vérifie chaque instruction par rapport à l'enregistrement du merge et où son inspection s'arrête. Le prix est que chaque écriture JIT dépend de la disponibilité du runner et de la gateway. Avec une gateway qui ne fait que distribuer le credential, le runner est une convention, pas une protection contre une instruction sans clause `WHERE`.

Où va le snapshot

Le snapshot avant est une copie de données, et il lui faut un responsable et une expiration comme à toute autre. Un snapshot des valeurs de `status` sur `invoices` est une donnée Internal. Un snapshot d'une colonne Confidential, comme `billing_address` sur `customers`, est Confidential, et le stockage a besoin de règles d'accès au moins aussi strictes que la table source.

Je conserverais les snapshots avec l'enregistrement de la correction, avec une durée de rétention fixée par l'équipe plateforme : assez longue pour pouvoir répondre à « cette correction était-elle juste ? », assez courte pour que le snapshot ne devienne pas une seconde copie, non gouvernée, des données clients. Cela représente quelques jours pour des status de factures et peut-être un cycle de facturation complet pour une adresse de facturation. Choisissez un nombre, notez-le, et laissez le snapshot expirer.

Le plan de rollback

Un snapshot avant n'est pas un plan de rollback tant que quelqu'un n'a pas écrit comment il serait utilisé. Le plan tient en une ou deux phrases dans l'en-tête de la correction, comme « restaurer `status` depuis le snapshot pour les mêmes clés », et il devrait être possible de transformer mécaniquement le snapshot en script de correction.

Le rollback est lui-même une correction de données. Il passe par la même demande et la même revue, parce que d'autres principals ont pu agir sur le nouvel état entre-temps et que restaurer les anciennes valeurs peut écraser des mises à jour légitimes. Le runner d'un rollback vérifie donc que les valeurs actuelles correspondent toujours à l'état après enregistré par la correction d'origine, et refuse de toucher aux lignes qui ont changé depuis. Une correction inverse qui ne sait pas dire « ces lignes ont évolué, traitez-les à part » provoque la réparation suivante.

Certains changements n'ont pas de rollback, et l'en-tête devrait le dire. Un e-mail, un webhook ou un appel à un prestataire de paiement ne se reprennent pas en restaurant une colonne. Pour ceux-là, la conversation a lieu avant l'exécution de la correction : quels effets ce changement d'état déclenche-t-il, et faut-il d'abord mettre en pause les services qui réagissent ?

Deux tables n'ont pas de rollback au sens ordinaire, parce qu'aucun rôle humain ne peut les modifier. `ledger_entries` et `audit_events` sont append-only. Une correction du registre est une nouvelle entrée, émise uniquement par le chemin de correction de `payments-service`, jamais par un login humain ; une correction qui semble nécessiter un changement à cet endroit est donc une demande pour ce chemin, pas pour un grant. Un trigger qui rejette tout `UPDATE` et tout `DELETE` sur les deux tables fait tenir la règle même si un grant est élargi par erreur. Il a un coût : `ledgerly_owner` peut toujours supprimer le trigger, si bien que le véritable point de revue reste sur les migrations qui touchent ces tables (Chapitre 11).

Un DELETE suit la même procédure, avec un rollback plus difficile

L'anatomie ci-dessus est écrite pour `UPDATE`. Supposons qu'un retry dans `invoice-worker` ait inséré des lignes en double dans `invoice_lines`, et que la réparation les supprime. `DELETE` sur `invoice_lines` se trouve dans le plafond de `human_jit_write`, et il diffère sur trois points.

Le grant borne la table et non les lignes, parce qu'un triplet `DELETE` n'a pas de colonnes ; la borne vient donc du filtre, du décompte attendu et de la revue, et le filtre devrait être une liste de clés. Le snapshot est la ligne entière, puisque rien d'autre ne conservera les valeurs ensuite, et une table avec des colonnes Confidential rend le snapshot aussi sensible que la table. Le rollback est une réinsertion, qui est une autre correction de données avec sa propre demande. Un triplet `INSERT` nomme ses colonnes ; la demande de réinsertion décide donc si `id` et `created_at` en font partie. Si ce n'est pas le cas, la ligne restaurée reçoit de nouvelles valeurs pour les deux, et tout ce qui référençait les anciens identifiants n'est pas restauré. Si c'est le cas, l'approbateur voit écrire une clé et un horodatage, ce que ne fait aucune correction ordinaire, et ce que le moteur fait d'une clé fournie dépend de la manière dont il génère les clés. L'en-tête indique quel choix fait le rollback.

```
fix:
    id              "FIX-0044"
    intent          "Remove the duplicate invoice_lines rows created
by a retry.
                     No other row is deleted."
    target_filter   "id IN (<ids of the duplicates>)"
    change          "DELETE"
    expected_rows   12                          // illustrative
    snapshot        "whole rows"
    privileges      ("DELETE", "invoice_lines")  // no columns for
DELETE
    rollback        "re-insert from the snapshot as a new fix; its
INSERT triple
                     leaves out id and created_at, so the rows get new
values and
                     references to the old ids are lost"
    effects         "invoice totals are recomputed by the worker on
next run"
    watch_tables    ("invoices")
```

Là où la perte des identifiants n'est pas acceptable, le relecteur demande une protection moins coûteuse que le rollback : une clé étrangère qui rejette la suppression tant qu'une autre ligne pointe encore vers la ligne, ou une réparation différente qui ne supprime pas la ligne.

La revue: le principe des quatre yeux, dans un dépôt

Une correction de données est du code qui s'exécute une fois ; elle reçoit donc le contrôle auquel les ingénieurs font déjà confiance : une pull request. La correction est un script dans un dépôt, avec un en-tête, un relecteur et un historique, et l'historique répond, six mois plus tard, à la question de savoir qui a changé un status et pourquoi.

```
// data-fixes/2026-10-06-FIX-0031-reset-sending-invoices.fix
fix:
    id              "FIX-0031"
    intent          "Invoices stuck in 'sending' after the worker bug
                     are set to 'sent'. No other column changes."
    target_filter   "status = 'sending' AND updated_at < '<cutoff>'"
    change          "SET status = 'sent'"   // applied to
target_filter only
    expected_rows   40
    postcondition   "no invoice matches target_filter"
    privileges      ("UPDATE", "invoices", "status")
    rollback        "restore status from snapshot for the same ids,
                     only where status is still 'sent'"
    effects         "reminders resume for these invoices"
    watch_tables    ("invoices")            // read by the dry run
    author          "robin"
    reviewer        "alex"
```

Le champ `privileges` relie le dépôt au Chapitre 8. Les triplets de la demande en sont copiés, si bien que l'approbateur regarde la demande et la correction côte à côte, et `loadApprovedFix` refuse une demande dont les triplets diffèrent de la correction. Une instruction qui touche une colonne que ses privilèges ne nomment pas échoue à l'exécution, ce qui est le résultat voulu ; le grant est la seconde ligne de défense derrière la revue.

Ce que lit le relecteur

Le travail du relecteur est plus étroit que « est-ce une bonne idée ? ».

- **L'instruction correspond-elle à l'intention ?** Elles doivent concorder sur les tables, les colonnes et le filtre.
- **Le filtre est-il précis ?** Un filtre qui sélectionne par clé est le plus sûr ; un filtre qui sélectionne par état et par une date limite est acceptable si le décompte concorde ; un filtre sans aucune restriction est rejeté d'emblée.
- **Le décompte attendu est-il plausible, et d'où vient-il ?** Un relecteur qui voit `expected_rows: 40` peut demander la sortie de la requête de décompte dans la pull request. Un nombre dont personne ne peut indiquer la source est un nombre que personne n'a vérifié.
- **Les privilèges ne nomment-ils que ce dont l'instruction a besoin ?** Comparez-les ligne à ligne.
- **La correction touche-t-elle une colonne Restricted, une table append-only ou une table hors du plafond ?** Si oui, ce n'est pas une

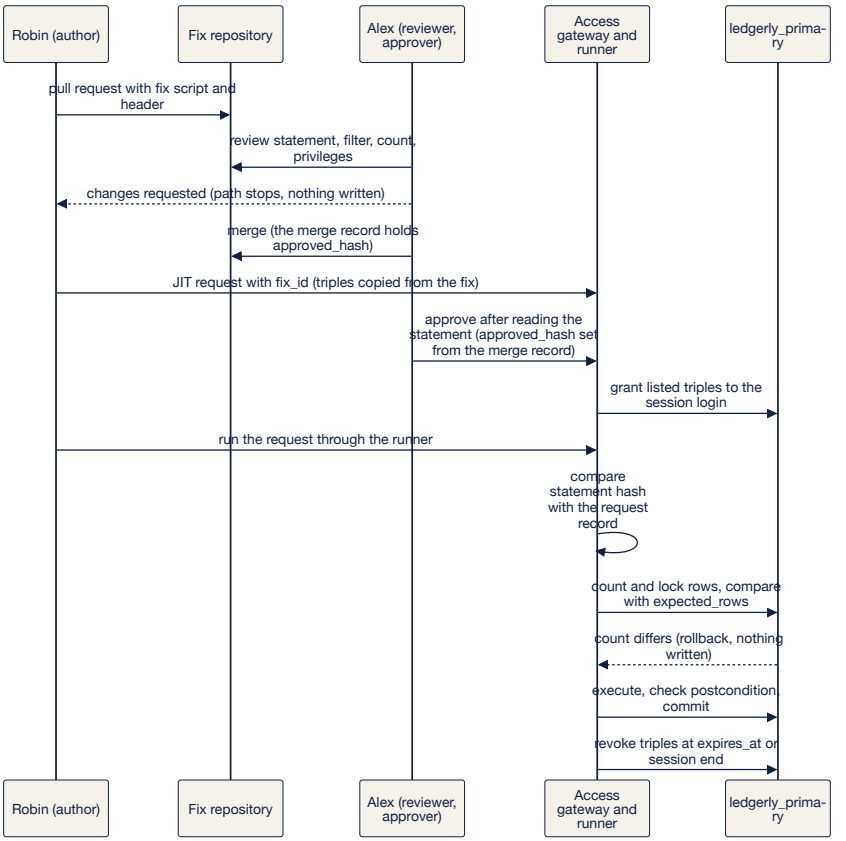
correction de données, et la revue se termine par un renvoi vers le bon chemin.

- **Que déclenche ce changement ?** Des relances, des e-mails, des jobs qui lisent la colonne. Cette question attrape les cas discrets.
- **L'aperçu concorde-t-il avec l'intention ?** La pull request contient la sortie de la requête de décompte et un aperçu `SELECT` des lignes et des valeurs après ; le rapport du dry-run est joint à la demande plus tard, et un trigger ou une cascade apparaît là et nulle part ailleurs.
- **La phrase de rollback est-elle utilisable ?** Pour un `DELETE`, elle dit ce que la réinsertion ne peut pas restaurer.

La revue applique le principe des quatre yeux au sens du livre : une seconde personne relit avant qu'une action risquée s'exécute. Le merge fige le texte : une modification du script après la revue produit un hash différent, et le runner la refuse. L'approbateur de la demande JIT et le relecteur de la correction accomplissent deux actes distincts. Dans une grande équipe, je préférerais deux personnes, afin que l'approbateur vérifie l'accès par rapport à la correction au lieu de la relire ; dans une petite équipe, ce sera souvent la même personne, ce qui est acceptable si ni l'une ni l'autre n'est l'auteur. L'auteur n'approuve jamais sa propre correction ni son propre grant.

La séquence

Le diagramme montre le chemin qui va d'une demande à un changement ayant fait l'objet d'un commit. Repérez les endroits où le chemin peut s'arrêter sans que rien ne soit écrit : la revue, la comparaison des hashs et les guards du runner.



Quand le changement porte sur une colonne Restricted

Le plafond interdit les colonnes Restricted dans toute demande JIT (le Chapitre 3 en donne la raison) ; toute une classe de réparations n'a donc pas de chemin SQL. Imaginez que l'`account_holder_name` d'un client sur `bank_accounts` ait été saisi avec une faute de frappe. Une personne ne peut pas le corriger dans une session JIT, parce que la colonne ne peut pas être nommée dans une demande. Une session break-glass ne le peut pas non plus : `human_breakglass` ne détient aucun privilège d'écriture sur les colonnes Restricted, quoi que dise le processus.

Le chemin est l'application. `admin-console` demande à `payments-service`, le seul à écrire dans `bank_accounts`, de faire le changement ; `admin-console` exige qu'une seconde personne l'approuve avant que `payments-service` ne l'effectue, et le changement est enregistré dans `audit_events`. C'est un chemin produit, pas une correction de données : il

a un formulaire, une validation, un champ de motif, un approbateur et un enregistrement, et il s'exécute avec le rôle propre du service, pas avec celui d'un humain.

Cela coûte plus cher qu'une instruction SQL, et je pense que c'est normal. Ces valeurs sont celles dont l'exposition nuit aux clients de manières difficiles à réparer, et l'application est le seul endroit où vivent la validation et l'ensemble complet des effets de bord. Le coût se compte en minutes pour la personne qui a besoin du changement ; le gain est qu'une réparation tapée à la main ne peut plus introduire un numéro de compte bancaire malformé. De même, un token erroné dans `payment_tokens` est remplacé par `payments-service` via le flux propre au prestataire, et non rapiécé.

Le break-glass n'est pas une voie de correction de données

`human_breakglass` écrit là où le plafond JIT ne va pas, et c'est le seul rôle humain qui lit les colonnes Restricted (ses privilèges exacts figurent au Chapitre 3). Sur le papier, il ressemble à un moyen commode de faire une réparation que le chemin JIT refuse. Ce n'en est pas un, et la raison tient à ce qui manque au rôle.

Le break-glass est auto-déclaré. Il n'a ni approbateur, ni `fix_id`, ni runner, et ses garde-fous sont l'ensemble de logging et la revue obligatoire que définit le Chapitre 8. Une action de limitation qui arrête un dommage en cours, relue après coup, est ce pour quoi le rôle existe (le Chapitre 15 couvre le versant incident). La réparation durable des données est un autre acte et c'est toujours une correction de données : exécutée dans une session break-glass, elle saute le seul contrôle qui lit l'instruction avant son exécution, dans un rôle où un `UPDATE` sans row guard est possible. Le Chapitre 8 l'énonce comme un risque accepté, et faire passer les réparations par ce rôle en ferait une routine.

La seconde raison tient aux incitations. Si le break-glass peut faire une réparation, il devient la lane rapide, et son alerte cesse de signifier quoi que ce soit. Une correction de données qui « a besoin » du break-glass signale autre chose : le cycle d'approbation de nuit est trop lent, le plafond n'inclut pas une colonne dont la correction a légitimement besoin, ou la correction relève du chemin applicatif. Si une réparation ne peut pas attendre, arrêtez le dommage avec la plus petite action disponible et exécutez la correction de données par le chemin normal une fois la pression retombée.

Transformer une correction répétée en fonctionnalité produit

Une correction de données est un symptôme : un état est atteignable que l'application ne peut pas défaire. La première occurrence est une correction ; la troisième est une demande de fonctionnalité que personne n'a déposée. Je conserverais un répertoire de scripts de correction, étiquetés par type de réparation, et quand la même étiquette réapparaît, quelqu'un demande si elle a sa place dans `admin-console`. « Trois fois » n'a rien de précis : une réparation de dix lignes deux fois par an ne justifiera peut-être jamais un écran, tandis qu'une réparation qui s'exécute chaque semaine, demande une revue à chaque fois et pourrait être faite par le support en une minute est une fonctionnalité qui attend d'être écrite.

Reprenez les factures bloquées du début, et supposons que l'état se reproduise. La table des grants du Chapitre 3 montre déjà ce que `app_admin_console` peut faire sur `invoices` : `SELECT`, et `UPDATE` sur `status` uniquement. La version produit est une action sur la page de la facture, accessible à un groupe restreint, qui réinitialise le status d'une facture, demande un motif et écrit l'action dans `audit_events` avec l'identité du principal humain. Elle porte ses propres vérifications : uniquement les factures dans l'état bloqué, une seule à la fois, et une limite au nombre qu'une personne peut réinitialiser en une heure.

La conversion déplace le risque de l'instruction, qui paie une demande, une revue et un runner à chaque réparation, vers le code, qui est relu, testé et versionné une seule fois. Le support peut utiliser l'action sans session de base de données ; chaque réparation qui devient une fonctionnalité est donc une écriture récurrente qui n'a plus besoin d'accès humain.

```
function resetStuckInvoice(actor, invoice_id, reason):
    if not actor.canResetInvoices:
        raise NotAuthorized
    if reason is empty:
        raise ReasonRequired
    if countResetsBy(actor, within: "1 hour") >= resetLimit:
        raise RateLimited

    transaction:
        invoice = readInvoiceLocked(invoice_id)
        if invoice.status != "sending" or invoice.updated_at >
cutoffForStuck():
            rollback()
            raise NotStuck
        setInvoiceStatus(invoice_id, "sent")
        // commit point: the status change and the business record
agree
        appendAuditEvent({ actor_principal: actor.id, action: "reset-
stuck-invoice",
                           subject_id: invoice_id, reason: reason })
        commit()
```

L'événement d'audit et le changement de status se trouvent tous deux dans `ledgerly_primary`, si bien qu'une seule transaction les couvre ; `audit_events` est l'enregistrement métier de l'application, pas le log d'audit du Chapitre 14. Dans le runner ci-dessus, le log d'audit est un système séparé, et c'est l'ordre des opérations qui porte la sécurité.

Toute réparation ne doit pas devenir une fonctionnalité. Un bouton sur lequel n'importe qui peut appuyer a un profil de risque différent de celui d'un script que deux personnes ont relu ; les réparations qui touchent à l'argent, aux tokens ou aux coordonnées bancaires restent donc sur un chemin soumis au principe des quatre yeux, même lorsqu'elles sont fréquentes. Une réparation ne devient une action en libre-service que si son pire cas est borné par la conception de l'action ; sinon, elle devient une demande avec une approbation à l'intérieur du produit. Le test consiste à savoir si la réparation va se reproduire et si une personne est actuellement la seule à pouvoir la faire. Si les deux réponses sont oui, le sprint est un investissement et non un coût.

Des runbooks pour les réparations prévisibles

Certaines corrections se reproduisent par type même quand les détails changent : réattribuer une facture au bon client, rejouer un export en échec, réparer une ligne dupliquée. Pour elles, un **runbook** est l'artefact adapté : une procédure écrite et suivie pour une tâche répétable. Il a le même squelette qu'un script de correction, avec les paramètres laissés vides. Il nomme le symptôme qui le déclenche, les requêtes de diagnostic qui le confirment, le modèle de la correction et les privilèges nécessaires, la source du décompte attendu, les effets à vérifier, le rollback, et qui peut écrire et relire. Il dit aussi quand s'arrêter et faire remonter : « si le décompte du diagnostic dépasse le nombre indiqué dans ce paragraphe, il ne s'agit pas du problème connu ».

Un runbook peut aussi porter une pré-approbation, afin qu'une réparation prévisible n'attende pas un approbateur la nuit. Cinq conditions en décident, et c'est le runner qui les vérifie, pas la personne :

- le filtre sélectionne par clé, une liste d'ids, et non par un état et une date limite ;
- le nombre de lignes attendu est d'au plus N, un nombre que le runbook fixe pour sa propre réparation (disons 20 pour les factures bloquées ; à titre d'illustration) ;
- les triplets sont exactement l'ensemble fixe du runbook ;
- la réparation ne déclenche aucun effet sortant, comme l'enregistre le champ `effects` du runbook ;
- l'instruction est le modèle du runbook dans lequel seule la liste de clés a été substituée, et le hash du modèle a été approuvé lors de la revue du runbook.

L'enregistrement du merge du runbook stocke ce hash de modèle. La liste de clés est la seule partie que porte la demande, et le runner la substitue dans le modèle stocké et vérifie le résultat par rapport aux triplets de la demande ; il ne prend jamais un hash ni une instruction de son propre appelant (le Chapitre 9 explique comment la gateway vérifie l'instruction substituée). Si une condition échoue, la demande retombe dans la revue complète avec un approbateur.

Le coût est qu'une exécution pré-approuvée s'exécute sans que personne ne lise la liste de clés cette nuit-là ; la limite de lignes du runbook borne donc une liste erronée, et une personne autre que celle qui a lancé l'exécution lit l'enregistrement le jour ouvré suivant. Un runbook vieillit aussi : un runbook pour une table qui a changé de forme depuis donne à une instruction périmée l'apparence d'une revue. Associez chaque runbook à une date et à un responsable, et traitez une exécution en échec comme une invitation à le réviser, non comme un désagrément à contourner.

Un UPDATE sans WHERE, analysé sans blâme

Prenons Ledgerly dans son état initial. Un ingénieur se connecte avec un client de bureau qui mémorise le mot de passe admin partagé et fait un commit automatique. La tâche concerne les factures bloquées. L'instruction est écrite sur deux lignes dans l'éditeur : l'`UPDATE` sur la première, le `WHERE` sur la seconde. Le client exécute la sélection en surbrillance, et la sélection ne couvre que la première ligne. L'instruction s'exécute, et chaque ligne de `invoices`, disons neuf mille, a le status `sent`.

Une version de cette histoire se termine par quelqu'un à qui l'on dit d'être plus prudent. Elle n'apprend rien, parce que le prochain ingénieur à la même place fait face à la même distance entre une question et un changement irréversible. La question utile est de savoir ce qui, dans la conception, a permis à une seule instruction d'atteindre toutes les lignes.

- **Le login pouvait écrire partout.** Le rôle admin partagé est propriétaire de toutes les tables, si bien qu'une erreur n'était pas confinée à la table ou à la colonne dont la tâche avait besoin.
- **Le client faisait le commit de lui-même.** Personne n'a vu « neuf mille lignes affectées » avant que l'instruction ne devienne définitive.
- **Le décompte n'a jamais été demandé.** Rien n'obligeait la personne à savoir combien de lignes attendre, si bien que le nombre affiché après coup n'avait aucun sens.
- **Personne d'autre n'avait lu l'instruction.** Le seul lecteur était l'auteur, qui relisait l'intention dans le texte.
- **Il n'y avait pas de snapshot avant.** Les anciens status avaient disparu, et la réparation de la réparation n'avait pas de point de départ.

- **Le log disait « le rôle partagé ».** La question « qui a exécuté ceci ? » n'avait de réponse qu'à travers l'historique du chat.

La reprise part d'une sauvegarde, réconcilie tout ce qui s'est passé depuis et informe tous ceux qui sont en aval (le Chapitre 13 et le Chapitre 15 traitent de ces mécanismes). La conception en avait fixé le coût avant que l'instruction ne soit tapée.

Faites maintenant passer le même après-midi par l'état cible. Robin écrit la correction avec son en-tête et un décompte attendu de 40, et Alex la relit. La demande JIT nomme `UPDATE` sur `invoices`, colonne `status`, et le grant dure une heure. Si le filtre avait disparu du script relu, le runner compterait neuf mille lignes contre 40 attendues et refuserait avant qu'aucune instruction ne s'exécute. Une instruction tapée dans le client personnel de la personne ne serait pas transmise comme écriture (la règle de gateway du Chapitre 9), et dans une simple session `human_readonly`, l'`UPDATE` ne s'exécuterait pas du tout.

Le guard ne rend pas une correction juste. Il rend bruyante une classe d'erreurs : l'instruction a touché un nombre de lignes différent de celui qu'avait annoncé l'auteur. Une instruction qui touche exactement 40 lignes, mais les 40 mauvaises, passe ; c'est pourquoi la revue, la postcondition et le snapshot existent comme couches séparées, et pourquoi le Chapitre 9 dit ce que l'analyse ne peut pas voir. Le guard ne vaut aussi que ce que vaut son nombre. Un décompte attendu copié depuis la sortie de l'exécution elle-même apprend au runner à être d'accord avec lui-même ; le nombre doit donc venir d'une requête que le relecteur peut exécuter indépendamment, idéalement avant que l'instruction n'ait été écrite. Là où le moteur dispose de grants au niveau des colonnes (Chapitre 3), un grant étroit limite ce que peuvent atteindre les cas que le guard manque.

Compromis et coûts

La procédure est plus lente que taper une instruction. Une demande, une revue, une approbation et un runner ajoutent des minutes un bon jour, et beaucoup plus quand l'approbateur dort. Un jeudi après-midi, c'est acceptable. La nuit, sous une alerte, c'est le coût qui pousse les gens vers le break-glass, et il mérite une attention de conception : un relecteur dans la rotation, la pré-approbation par runbook décrite plus haut pour les ré-

parations qui s'y prêtent, et une limite explicite au temps maximal que l'équipe attendra une approbation de nuit, écrite avant la première nuit où elle sera nécessaire.

La revue peut devenir un réflexe. Une approbation donnée en quelques secondes ajoute de la friction sans réduire l'exposition ; le Chapitre 8 développe l'argument complet. Le signal propre à ce chapitre est la pull request : si les relecteurs ne demandent jamais la sortie de la requête de décompte et ne demandent jamais de modification d'une instruction, ils ne lisent rien ou ne relisent que des corrections parfaites. Les défenses sont les points de checklist qui exigent un artefact : la sortie du décompte et l'aperçu dans la pull request, le rapport du dry-run sur la demande, et une comparaison explicite de l'instruction avec les privilèges.

Les transactions détiennent des verrous. Le runner démarre la transaction après la revue et la termine en quelques secondes. Une procédure qui l'ouvre d'abord, « par sécurité », et laisse l'humain décider ensuite fait attendre les autres écrivains, remplit les connexions du pooler et fait se déclencher les timeouts du Chapitre 5 sur un travail sans rapport. Gardez la décision humaine hors de la transaction.

Les grosses corrections ont besoin de lots. Une correction qui touche des millions de lignes ne peut pas s'exécuter en une seule transaction sans verrous détenus longtemps et sans un gros journal de transactions. La réponse habituelle est de la découper en lots, disons quelques milliers de lignes chacun (une taille illustrative, pas une recommandation), chacun avec son propre décompte et son propre guard, avec un mode lot du runner qui répète les mêmes vérifications. Une défaillance à mi-chemin laisse les données dans un état mixte ; la correction doit donc être écrite pour pouvoir être réexécutée : le filtre ne sélectionne que les lignes encore dans l'ancien état. Une grosse réparation reste une correction de données ; seul le premier remplissage d'une nouvelle structure relève d'une migration (Chapitre 11).

Les snapshots et le dépôt de corrections ont besoin d'un responsable. Un snapshot sans expiration a échangé un risque contre un autre, et pour les colonnes Confidential, le second peut être le plus grand. Un répertoire de scripts de correction est un enregistrement de la manière dont la production a dérivé, et sans index par étiquette, personne n'ira chercher l'occurrence précédente d'un problème.

Quand ne pas utiliser tout cela. Dans une petite équipe dont la base de données ne contient rien de sensible, une pull request pour chaque changement de status est plus de cérémonie que le risque ne le justifie ; compter d'abord et utiliser une transaction restent rentables. Sur une base de développement ou de staging sans données de production, rien de tout cela ne s'applique, et pour une copie reconstruite chaque nuit, comme `ledgerly_analytics`, une écriture erronée coûte une reconstruction. La procédure s'adapte au coût d'une erreur, et c'est sur le primary que ce coût est le plus élevé.

Erreurs courantes

- **Traiter la clause `WHERE` comme le guard.** C'est tentant parce que la clause est l'endroit où vit l'intention de l'auteur. Mais une clause est du texte, et le texte peut être coupé, mal tapé ou exécuté en partie. Le remède est un guard qui compare le décompte modifié avec une attente déclarée et fait un rollback en cas d'écart.
- **Compter sur la réplique et mettre à jour sur le primary.** La réplique a du retard, donc les deux nombres peuvent différer pour une bonne raison. Le remède est de prendre le décompte faisant foi sur le primary, dans la transaction, et de n'utiliser le décompte de la réplique que pour explorer.
- **Retaper l'instruction relue.** Le relecteur a approuvé un fichier, et une personne l'a copié dans un éditeur et a changé une chose « pendant qu'elle y était ». Le remède est un runner qui compare le hash du script avec l'`approved_hash` de l'enregistrement de la demande, avec une gateway qui ne transmet les écritures de la session que depuis ce runner. Sur une gateway qui ne fait que distribuer des credentials, le guard est une convention, et c'est la revue qui doit le porter.
- **Demander un grant large « par sécurité ».** Cela fait de l'approbation un jugement de confiance au lieu d'une vérification par rapport à une instruction, et cela élargit ce qu'une erreur peut atteindre. Le remède est de demander les triplets dont l'instruction a besoin, et de faire une seconde demande si la situation change.
- **Utiliser le break-glass parce que l'approbateur est lent.** Le chemin est plus court et la correction est urgente. Le remède est de résoudre

la lenteur de l'approbation de nuit et de garder le break-glass pour les urgences pour lesquelles il a été défini.

- **Écrire une colonne Restricted via une session « spéciale ».** Mettre à jour un `iban` directement ressemble à un service rendu au client, et cela supprime d'un coup la validation, l'approbation et l'enregistrement de l'application. Le remède est le chemin applicatif, même quand il est plus lent.
- **Enregistrer un snapshot que personne ne gouverne.** Le snapshot protège contre une mauvaise correction et devient une copie non gouvernée de données Confidential dans un dossier partagé. Le remède est un responsable, une règle d'accès alignée sur la table source et une date d'expiration.
- **Sauter la phrase de rollback parce que la correction est petite.** Les petites corrections sont celles qu'on annule dans la précipitation. Le remède est un plan d'une ligne, écrit avant l'exécution, qui dit ce que le snapshot restaure et où il ne doit pas le faire.
- **Laisser une correction répétée à l'état de script pour toujours.** Le résultat est une écriture récurrente qui exige à chaque fois un humain sur le primary. Le remède est un examen périodique des étiquettes du répertoire de corrections, et une décision, même si la décision est « pas encore ».
- **Relire la justesse de l'intention au lieu de la justesse de l'instruction.** Un relecteur qui débat de l'intérêt de la réparation peut ne jamais lire l'instruction. Le remède est une checklist qui pose les questions étroites dans l'ordre.

Points clés

- Une session en lecture seule est le bon défaut pour l'investigation, et elle ne fonctionne que si la tâche courante ne demande aucune approbation.
- Écrire est un autre acte que lire ; l'écriture passe donc par sa propre demande, sa propre revue, sa propre limite de temps et son propre enregistrement. La demande porte un `fix_id` et un `approved_hash` tiré de l'enregistrement du merge, ses triplets sont dérivés de la correction, et l'approbateur voit l'instruction.

- L'ordre sûr d'une correction de données est : décompte, snapshot, transaction, guard sur le nombre de lignes, postcondition, commit ; et c'est le runner, non la mémoire de la personne, qui devrait l'imposer, avec une gateway qui ne transmet les écritures de la session que depuis le runner.
- Un guard rend bruyante une classe d'erreurs et ne rend pas une correction juste ; c'est pourquoi la revue, les grants étroits et un snapshot avant existent comme couches séparées.
- Un changement sur une colonne Restricted n'est pas une correction de données : il passe par `admin-console` vers `payments-service` avec un second approbateur et un enregistrement dans `audit_events`, et le break-glass n'est pas un moyen de le contourner.
- Une correction manuelle répétée est une fonctionnalité produit manquante, et chacune qui devient une action de `admin-console` est une écriture récurrente qui n'a plus besoin d'une personne sur le primary.
- Quand une instruction atteint toutes les lignes, la question utile est de savoir ce qui, dans la conception, l'a permis, et la réponse est une liste de guards manquants, pas une personne.

Les corrections de données changent ce qui se trouve dans les tables ; le type de changement suivant modifie les tables elles-mêmes. Le Chapitre 11 se tourne vers les changements de schéma, vers qui les exécute et avec quelle identité, et vers l'habitude de les rendre compatibles vers l'arrière afin que la réparation d'un jour ne soit pas la panne du lendemain.

Chapitre 11 — Changements de schéma et accès pour les migrations

Imaginez un jeudi après-midi chez Ledgerly. L'équipe facturation veut une nouvelle colonne sur `invoices` afin que la taxe puisse être affichée séparément du total, et le changement est petit : une colonne, une valeur par défaut, un backfill pour les anciennes lignes. Robin, d'astreinte cette semaine, relit la pull request et l'approuve. Le pipeline exécute la migration en tant que `ledgerly_migrator`, le déploiement se termine, et pendant environ quatre-vingt-dix secondes, chaque requête vers `billing-api` qui touche une facture reste bloquée, puis échoue.

Rien n'était faux dans la migration, et personne n'a rien fait d'inhabituel. Une longue transaction de lot de `invoice-worker`, laissée ouverte pendant qu'elle attendait un service de messagerie externe, détenait un verrou sur la table ; la migration s'est mise en file derrière elle, et chaque instruction arrivée après la migration s'est mise en file derrière la migration. Alex, de l'équipe plateforme, trouve vite la cause. Vient ensuite la question plus difficile : la migration s'est exécutée avec le pouvoir du propriétaire de toutes les tables, elle s'est exécutée sans surveillance, et la seule chose entre ce pouvoir et le registre était la revue d'un diff. Comment les changements de schéma devraient-ils atteindre le primary, avec quelle identité, et comment éviter qu'ils deviennent soit un trou dans le modèle d'accès, soit une panne ?

Ce chapitre ajoute une étape de l'état cible : les migrations s'exécutent en tant que `ledgerly_migrator` depuis le pipeline, et rien d'autre n'a le pouvoir de modifier le schéma.

Dans le modèle en cinq parties du Chapitre 1, une migration relève de l'**Exécution**, avec la borne la plus large qui soit. Le pipeline détient les privilèges du propriétaire par conception ; le risque n'est donc pas dans la question de savoir qui détient un credential. Il est dans ce que la migration

est autorisée à faire, dans la manière dont ses instructions s'exécutent, et dans l'enregistrement qui subsiste. Pour les migrations, cela signifie une courte liste de classes d'instructions autorisées sur chaque lane (autorisation), un chemin qui décide quand et comment une instruction s'exécute, avec un lock wait court et un retry (exécution), et un enregistrement que le propriétaire ne peut pas réécrire (imputabilité).

Le problème, et pourquoi il est difficile

Un changement de schéma est l'opération la plus privilégiée que la plupart des équipes effectuent sur une base de données un jour ordinaire. La propriété permet au propriétaire de supprimer une table, de changer le type d'une colonne, de réécrire les grants et, sur de nombreux moteurs, de contourner les restrictions qui s'appliquent à tous les autres. Le Chapitre 3 a donné ce pouvoir à `ledgerly_owner`, un rôle sans login, et a permis à `ledgerly_migrator` de l'endosser uniquement pendant l'exécution d'une migration. Cet arrangement répond à la question de savoir qui est propriétaire du schéma. Il ne dit pas qui peut lancer une migration, ce qu'une migration en cours a le droit de faire, ni ce qui arrive à tous les autres pendant qu'elle s'exécute.

Trois risques distincts se trouvent dans cette seule activité.

Le premier est l'**autorité**. Quiconque peut déclencher l'étape de migration du pipeline peut, de fait, exécuter des instructions de niveau propriétaire sur la production. Ce qui est permanent, c'est l'autorité, au sens décrit au Chapitre 3 : une autorité de propriétaire permanente, exercée au moyen d'un credential de courte durée. Ses bornes sont procédurales : le credential ne vit que dans le pipeline, il expire avec le job, et le pipeline est le seul moyen sanctionné de l'utiliser. Ces bornes ne valent que ce que valent les contrôles d'accès du pipeline lui-même, ce qui fait du pipeline une partie de la frontière de sécurité de la base de données, que quelqu'un l'ait dessinée ainsi ou non.

Le deuxième est la **disponibilité**. Beaucoup de changements de schéma ont besoin de verrous, et c'est au verrou que les dégâts se produisent. Une instruction qui doit attendre un verrou ne fait pas qu'attendre : sur de nombreux moteurs, les instructions qui arrivent derrière elle se mettent aussi en file ; une migration qui n'obtient pas son verrou dans la première

seconde peut donc fermer la table à tout le monde aussi longtemps que la plus ancienne transaction devant elle reste ouverte. La taille du changement ne compte presque pas. Un changement d'une colonne peut être aussi perturbateur qu'une réécriture si la mauvaise transaction est sur le chemin.

Le troisième est la **dérive**. Un schéma défini par des migrations et aussi modifié à la main est un schéma que personne ne peut décrire entièrement. Un index ajouté en hotfix depuis une console un soir difficile est utile, et un an plus tard, c'est une différence entre environnements dont personne ne se souvient, accompagnée de grants qui ne correspondent à aucun fichier.

Chacun peut être réduit par la conception. Un rôle de migration borné par le pipeline, un lock wait qui échoue vite, une forme de changement qui n'a jamais besoin d'un long verrou, et une vérification qui compare le schéma réel au schéma attendu sont autant de choses que l'on peut construire. Une politique qui dit « soyez prudents avec les migrations » ne figure pas sur cette liste.

Qui exécute le DDL, et avec quel rôle

Le **DDL** (data definition language) est la famille d'instructions qui créent, modifient et suppriment des objets de schéma, par opposition aux instructions qui lisent et écrivent des lignes. La règle de ce chapitre est courte : le DDL s'exécute en tant que `ledgerly_migrator`, depuis le pipeline, et aucun rôle applicatif ne peut l'exécuter.

Les rôles applicatifs ne détiennent aucun DDL

Aucun des rôles `app_*` du Chapitre 3 n'est propriétaire d'une table ni ne peut en créer, en modifier ou en supprimer une. C'est facile à dire et un peu coûteux à tenir, parce que de nombreux frameworks peuvent créer ou modifier des tables au démarrage, et que certaines équipes l'activent parce que cela retire une étape du déploiement. Le prix de cette commodité est que le service a alors besoin de la propriété, et qu'un service compromis ou dérouté peut supprimer et réattribuer des grants. Je désactiverais la

gestion automatique du schéma en production, et je la garderais pour les bases de développement, où personne n'a à expliquer une table supprimée.

L'effet visible est qu'un déploiement qui inclut un changement de schéma comporte désormais deux étapes : la migration, puis le déploiement du service. L'ordre entre les deux est une question de conception à laquelle répond le pattern expand and contract décrit plus bas.

Le migrator et le propriétaire

`ledgerly_migrator` a un login qu'utilise le pipeline ; il est membre de `ledgerly_owner` et ne détient aucun privilège sur les données en propre. L'appartenance donne au migrator les privilèges du propriétaire, mais elle ne fait pas du migrator le propriétaire. Sur de nombreux moteurs, un nouvel objet appartient au rôle actif au moment de sa création ; une migration qui s'exécute sous le login propre du migrator laisserait donc ses tables appartenir au migrator. La règle est par conséquent la même sur tous les moteurs : une migration passe toujours explicitement à `ledgerly_owner`, parce que la propriété suit le rôle actif et non les privilèges hérités. Chaque objet qu'elle crée appartient alors à `ledgerly_owner`, et la migration revient au rôle propre du migrator quand elle se termine. Dans une notation qui n'appartient à aucun moteur :

```
function runMigration(migration):
    login = fetchMigratorLogin(identity: pipeline.workloadIdentity)
    connection = connect(login: login, database: "ledgerly_primary")
    // lock_wait 3s, statement timeout none (Chapter 5)
    applyRoleSettings(connection, role: "ledgerly_migrator")
    if isApplied(connection, migration.id):
        close(connection)
        return                          // a retry never re-runs
applied work
    // external record first: the owner can rewrite the database's own
record
    auditLogAppend({ actor: login.name, action: "migration-start",
                     migration: migration.id, commit: migration.commit
})

    try:
        assumeRole(connection, "ledgerly_owner")
        execute(connection, migration.statements)
        recordApplied(connection, migration.id)
    on LockWaitExceeded or DatabaseError as error:
        // owner rights end with the connection on every path
        resetRole(connection)
        close(connection)
        auditLogAppend({ actor: login.name, action: "migration-
failed",
                         migration: migration.id, error: error.kind })
        raise error
    resetRole(connection)
    close(connection)
    auditLogAppend({ actor: login.name, action: "migration-applied",
                     migration: migration.id })
```

Quatre détails comptent. Le credential est délivré à ce job par la workload identity du pipeline au moment de l'exécution (le pattern du Chapitre 4) et expire avec le job ; une variable de pipeline copiée ne peut donc pas détenir pendant des mois un credential de propriétaire valide, et rien dans le dépôt n'en détient un. Le passage au rôle propriétaire est limité aux instructions d'une seule migration. L'enregistrement indiquant qu'une migration a été appliquée est écrit dans la même exécution ; la base de données connaît donc sa propre version, ce dont dépend la vérification de dérive plus bas. Et l'entrée externe est écrite avant la première instruction de propriétaire, parce que `recordApplied` vit dans une base de données que le propriétaire peut réécrire. Le coût est qu'un stockage d'audit indisponible arrête la migration avant qu'elle ne commence. C'est l'ordre inverse de celui de la correction de données du Chapitre 10, où le log de session de la gateway enregistre déjà les instructions : ici, rien d'autre que le pipeline ne voit ce que fait la session du propriétaire, donc l'enregistrement passe en

premier. Chaque entrée de début reçoit une fin correspondante, `migration-failed` ou `migration-applied`, et `isApplied` rend un retry sûr même quand l'append d'audit final est l'étape qui a échoué.

Sur certains moteurs, l'enregistrement de la version et l'exécution du DDL peuvent partager une transaction ; sur d'autres, une instruction DDL termine la transaction qui l'entoure. Sachez lequel vous avez. Si une défaillance peut laisser le schéma modifié et la version non enregistrée, la migration doit être écrite de façon qu'une seconde exécution soit sans danger.

Les humains ne détiennent pas le migrator

Les humains ne se connectent jamais en tant que `ledgerly_migrator`. Si une personne a besoin d'un changement de schéma, elle écrit une migration, et le pipeline l'exécute. Cela ressemble à de la friction, et c'en est, mais cela procure une propriété utile : chaque usage du pouvoir du propriétaire correspond à une exécution du pipeline, qui a un commit, une revue et un log. Quand le credential du migrator se trouve aussi dans le gestionnaire de mots de passe d'une équipe, cette propriété disparaît, et avec elle la raison d'avoir un migrator.

Ce que le pipeline peut atteindre

Donner au pipeline le credential du migrator place un grand pouvoir dans un système que beaucoup de personnes peuvent influencer. Il vaut la peine de le dire explicitement, parce que c'est là que cette conception peut dérailler sans bruit.

Ce que le pipeline peut faire

Un job qui exécute `runMigration` peut créer, modifier ou supprimer n'importe quel objet du schéma, changer n'importe quel grant sur les objets dont le propriétaire est propriétaire, et modifier les deux tables append-only, `ledger_entries` et `audit_events`. Aucun rôle applicatif et aucun rôle humain ne peut modifier des lignes de ces tables, `human_breakglass` compris ; la seule exception structurelle est que le propriétaire, et donc une migration en cours, peut les modifier. Le contrôle de cette exception n'est pas un privilège. C'est la revue.

La question à poser sur le pipeline est donc : qui peut lui faire exécuter quelque chose ? Cela inclut les personnes qui font un merge sur la branche de déploiement, celles qui peuvent modifier la définition du pipeline, celles qui peuvent ajouter une étape à un job qui détient le credential, et quiconque peut lire l'environnement du job pendant son exécution.

Borner la portée

Je bornerais cette portée de cinq manières, à peu près dans l'ordre de ce que chacune retire.

1. **Un seul job détient le credential.** Le job de migration de l'environnement de production est le seul endroit où le login du migrator peut être délivré, et il est délivré par l'identité du job, pas par une variable que chaque job peut lire. Les builds de branches non relues, les jobs de test et les pipelines de pull request ne le reçoivent jamais.
2. **La définition du pipeline est relue comme les migrations.** Un changement du job de migration, de ses permissions ou des règles de branche qui le déclenchent est un changement de la portée du propriétaire, et il a besoin du même relecteur que celui qui vérifierait une migration.
3. **Le déclencheur est une branche protégée.** Un merge sur cette branche exige la revue d'une personne qui n'est pas l'auteur ; c'est le contrôle de Ledgerly selon le principe des quatre yeux pour les changements de schéma. Pour les migrations qui touchent aux grants, à la propriété ou aux tables append-only, le relecteur doit venir de l'équipe plateforme.
4. **Le credential est de courte durée.** Un job qui échoue et laisse un log contenant un secret laisse un secret déjà expiré.
5. **Chaque exécution est enregistrée.** Le pipeline écrit ce qu'il a appliqué, avec quel login, depuis quel commit, dans un stockage qu'il ne peut pas réécrire, et il écrit l'entrée de début avant que la migration ne s'exécute. Le Chapitre 14 traite de l'endroit où vit ce stockage.

La revue elle-même peut être en partie mécanique. Une petite vérification peut lire la migration avant une personne et signaler les instructions qui méritent un examen plus attentif :

```
function classifyMigration(migration, lane):
    flags = []
    refusals = []
    for each statement in migration.statements:
        if statement.touches("ledger_entries") or
statement.touches("audit_events"):
            add(flags, "append-only table: platform review required")
        if statement.kind in ["GRANT", "REVOKE", "CHANGE OWNER",
                              "CHANGE DEFAULT PRIVILEGES", "CREATE
VIEW"]:
            add(flags, "changes access: platform review required")
        if statement.kind in ["CREATE TABLE", "ADD COLUMN"] and
           (statement.table is new or statement.table in
sensitiveTables) and
           not hasExplicitGrants(migration, statement.table):
            add(flags, "new structure without a grant decision")
        if statement.kind in ["INSERT", "UPDATE", "DELETE"]:
            add(flags, "data change in a migration: backfill rule
applies")
        if statement.kind in lockHeavyKinds:
            add(flags, "long lock: scheduled window and a named
watcher")
        if statement.kind in ["DROP TABLE", "DROP COLUMN"] and not
migration.isContractStep:
            add(flags, "destructive change outside a contract step")
        if lane == "expedited" and statement.kind not in
expeditedAllowlist:
            add(refusals, "not allowed in the expedited lane: use the
normal lane")
    return flags, refusals
```

Dans la lane normale, la vérification ne décide rien. Elle fait porter l'attention du relecteur sur les lignes qui changent l'accès, détiennent un verrou ou détruisent des données, et elle rend visible l'absence d'un grant. Une vérification qui bloque un merge à chaque signalement, sans moyen d'en accuser réception, sera contournée en moins d'un mois ; un signalement exige donc un approbateur nommé et n'interdit pas le changement. La lane accélérée est le seul endroit où la vérification refuse : une liste `refusals` non vide fait échouer le job avant son exécution, et aucun approbateur ne peut passer outre. La section suivante explique pourquoi.

La lane normale ne refuse rien d'emblée, et c'est un choix : une liste stricte d'instructions interdites serait une seconde politique à côté de la matrice deny, gate et allow du Chapitre 9. La suppression de `ledger_entries` est signalée deux fois, comme changement d'une table append-only et comme changement destructif, et l'équipe plateforme doit y dire oui.

Pourquoi pas une personne avec un ticket

Certaines équipes préfèrent qu'un DBA applique chaque migration à la main après approbation. Pour une petite base de données avec un ou deux changements de schéma par mois, ce peut être un choix raisonnable, et la question de la portée du pipeline disparaît. Le coût apparaît quand la personne est absente, quand le changement doit être appliqué à l'identique dans plusieurs environnements, ou quand l'étape manuelle s'écarte de ce qui a été relu. La faiblesse du pipeline est l'étendue de sa portée ; celle du chemin manuel est que rien n'impose que ce qui a été appliqué soit égal à ce qui a été approuvé. Je penche pour le pipeline avec les bornes ci-dessus, et je reviendrais sur ce choix s'il s'avérait impossible de garder courte la liste d'accès du pipeline lui-même.

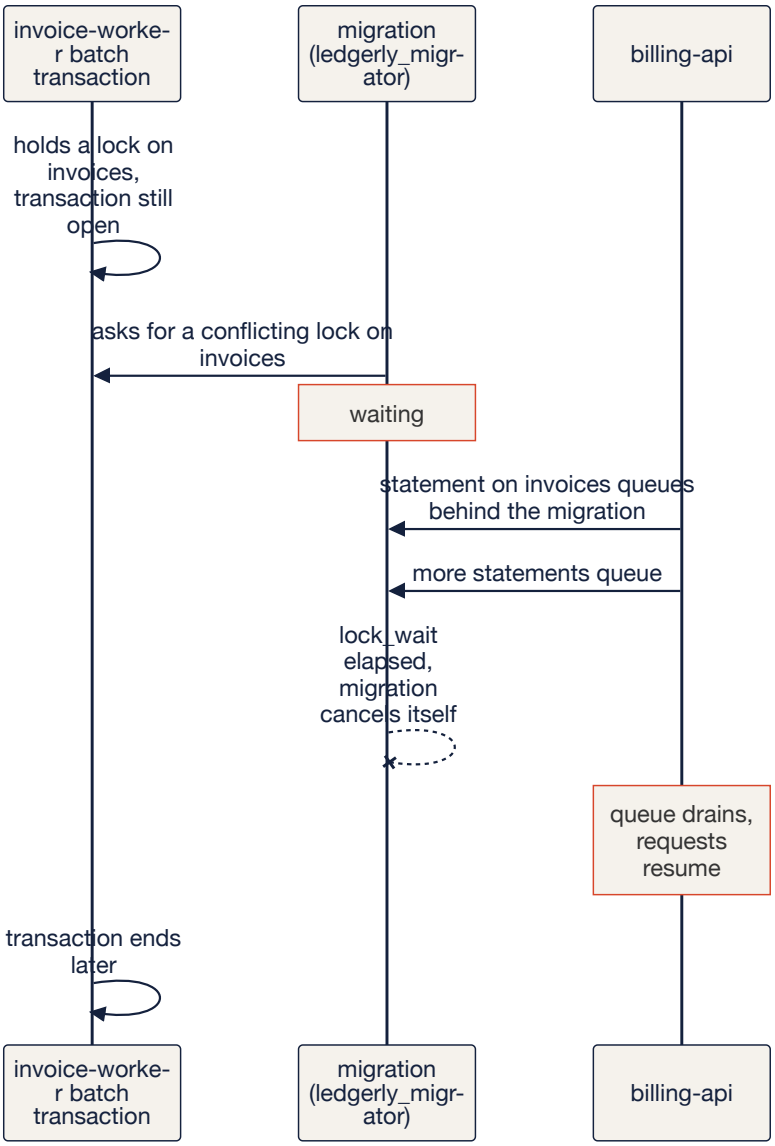
Les verrous et le risque pour la disponibilité

Revenons aux quatre-vingt-dix secondes de Ledgerly. Le mécanisme qui les explique est commun aux moteurs, et la défense se construit à partir de quelques habitudes.

Comment une migration en attente ferme une table

Une migration qui modifie `invoices` a besoin d'un verrou qui entre en conflit avec les lectures et écritures ordinaires. Elle ne peut prendre ce verrou que lorsqu'aucune transaction détenant un verrou en conflit n'est encore ouverte. Si une longue transaction est ouverte, la migration attend. Pendant qu'elle attend, les nouvelles instructions sur la table se mettent en file derrière la demande en attente de la migration, parce que les laisser passer devant l'affamerait. La table est alors de fait fermée jusqu'à ce que l'ancienne transaction se termine ou que la migration abandonne.

La séquence ci-dessous montre l'ordre des événements. Repérez le moment où le lock wait offre à la migration une sortie avant que la file ne grossisse.



Le point clé est l'auto-annulation. Sans lock wait, la migration attend aussi longtemps que cette transaction de lot reste ouverte, et `billing-api` reste en file pendant tout ce temps. Avec un lock wait court, la migration échoue au bout de quelques secondes, la file se vide et le pipeline peut ré-essayer plus tard.

Un lock wait court et un retry patient

Le Chapitre 5 a fixé `lock_wait 3s` et aucun timeout d'instruction pour `ledgerly_migrator`. Cette association est délibérée. Un changement de schéma peut avoir besoin de beaucoup de temps pour faire son travail, mais il devrait attendre très peu de temps pour commencer. Le retry appartient alors au pipeline :

```
function applyWithRetry(migration, maxAttempts, pauseSeconds):
    attempt = 1
    while attempt <= maxAttempts:
        try:
            runMigration(migration)
            return
        on LockWaitExceeded:
            // a clean no-op only if nothing had committed: one
transactional
            // unit, or a locking step that comes first
            if not migration.retrySafe:
                notify("platform", "partial migration, not retried: "
+ migration.id)
                raise MigrationBlocked
            notify("platform", "migration waiting on a lock: " +
migration.id)
            sleep(pauseSeconds * attempt)
            attempt = attempt + 1
    // the table is busy for longer than the retry window; a person
decides
    notify("platform", "migration blocked after retries: " +
migration.id)
    raise MigrationBlocked
```

Quelques remarques à ce sujet. Un lock wait en échec n'est un no-op propre que si rien n'a encore fait l'objet d'un commit ; c'est pourquoi une migration porte une marque `retrySafe` : elle forme une seule unité transactionnelle, ou son étape de verrouillage vient en premier. Une migration en plusieurs étapes, sur un moteur où chaque instruction DDL fait son propre commit, ne fait pas l'objet d'un retry aveugle, parce que la seconde tentative rencontrerait le travail de la première. Elle s'arrête et une personne décide. La pause augmente avec le nombre de tentatives, afin que les essais répétés ne deviennent pas eux-mêmes un motif de charge. Et l'état final, `MigrationBlocked`, est une question humaine : trouver quelle transaction gêne, et décider s'il faut attendre, l'annuler ou replanifier.

Cette décision implique de tuer la session de quelqu'un, ce qui est une question d'accès à part entière et relève du Chapitre 15 quand cela se produit pendant un incident.

Instructions longues et changements en ligne

Certaines opérations conservent un verrou pendant toute leur exécution, comme la réécriture d'une grande table pour changer le type d'une colonne. Pour celles-ci, le lock wait n'est pas le problème ; c'est la durée. Les moteurs diffèrent sur les instructions qui sont rapides, et la réponse change avec la version du moteur ; vérifiez donc chacune. Les outils de changement de schéma en ligne et les constructions d'index non bloquantes évitent le long verrou, mais ils créent et suppriment des objets ; ils ont donc besoin de la même portée de propriétaire que n'importe quelle migration et s'exécutent en tant que migrator, dans le pipeline. `classifyMigration` marque ces instructions comme à verrou long, et une migration à verrou long reçoit une fenêtre planifiée et une personne nommée pour la surveiller, inscrites dans la description de la migration. Ce n'est pas un échec de l'automatisation ; certains changements sont des événements.

Expand and contract

Le changement de schéma le plus sûr est celui avec lequel l'ancien et le nouveau code applicatif peuvent tous deux vivre. Le pattern correspondant est **expand and contract** : modifier le schéma par étapes additives, faire migrer l'application, et seulement ensuite retirer ce qui n'est plus utilisé. Il compte pour ce livre parce que chaque étape est une petite migration avec un petit verrou, et parce qu'il garde les déploiements réversibles.

Prenons un changement hypothétique chez Ledgerly, utilisé seulement pour en montrer la forme : renommer `total_minor` en `amount_minor` sur `invoices`. Le schéma canonique conserve `total_minor`, et l'exemple suppose une variante dans laquelle la colonne est renommée. Les étapes ressemblent à ceci :

1. **Expand.** Une migration ajoute `amount_minor` comme colonne nullable, avec un verrou bref. L'ancien code l'ignore.

2. **Écrire les deux.** Les nouvelles versions de `billing-api` et de `invoice-worker` écrivent la nouvelle colonne en plus de l'ancienne, si bien que les anciennes et les nouvelles versions peuvent tourner côte à côte.
3. **Backfill.** Les anciennes lignes reçoivent des valeurs pour la nouvelle colonne, comme décrit plus bas.
4. **Lire la nouvelle colonne.** Si la version se comporte mal, l'ancienne colonne est toujours juste ; le rollback est donc un déploiement et non une réparation.
5. **Contract.** Après une période pendant laquelle rien ne lit ni n'écrit `total_minor`, une dernière migration la supprime. C'est la seule étape destructive, et elle est soumise à un gate séparé : relue comme destructive, et jamais dans la version qui a introduit la nouvelle colonne.

Expand and contract coûte du temps et de l'attention. Un changement qui pourrait tenir en une migration en devient quatre ou cinq, réparties sur plusieurs déploiements, et pendant les étapes intermédiaires, l'application porte du code pour les deux formes. Pour une petite équipe avec un seul service et une fenêtre de maintenance, une migration unique peut être le meilleur choix. Pour Ledgerly, avec cinq services qui lisent les mêmes tables et aucune fenêtre que le métier accepte, c'est le pattern qui garde le verrou court et le rollback peu coûteux.

Les backfills dans la migration, et les corrections de données en dehors

L'étape 3 est un changement massif de données, et la même règle s'appliquerait à une séparation du total en montant net et taxe, avec une différence : l'ancienne table ne contient aucun chiffre de taxe ; ce backfill ne pourrait donc pas être calculé et devrait laisser les nouvelles colonnes vides jusqu'à ce qu'une correction de données les remplisse. Le Chapitre 10 définit une correction de données comme une modification des données de production faite par une personne en dehors du chemin normal de l'application, pour réparer un état que l'application ne peut pas réparer ; le livre a donc besoin d'une ligne de partage ici : un backfill qui appartient à un changement de schéma s'exécute dans la migration en tant que `ledgerly_migrator` ; tout autre changement massif de données est une correction de données et suit le Chapitre 10.

Qu'est-ce qui fait qu'un backfill appartient au changement de schéma ?

- Son seul but est de donner à la nouvelle structure ses premières valeurs, selon une règle énoncée dans la migration.
- Il est décrit dans la migration, relu avec elle et appliqué par le pipeline.
- Chaque valeur qu'il écrit pourrait être recalculée à partir de données déjà présentes dans la table.

Si l'une de ces conditions est fausse, c'est une correction de données. Le renommage montre où passe la ligne : chaque valeur écrite dans `amount_minor` est le `total_minor` de la même ligne, une règle que la migration énonce et qu'un relecteur peut vérifier. Supposons que l'équipe remarque, pendant le backfill, que certaines anciennes factures ont un `total_minor` faux. Corriger ces totaux ne fait pas partie du changement de schéma. Cela suit le Chapitre 10, avec un décompte des lignes concernées, un guard sur le nombre de lignes attendu et un second relecteur, parce qu'une copie qui préserve une valeur fausse reste un backfill correct. Le pouvoir du migrator est grand, et il ne devrait pas devenir la voie de tout changement délicat.

Sur une grande table, le backfill s'exécute par lots, chacun étant une courte transaction sur un ensemble borné de lignes. Il fait une pause quand le retard de la réplique augmente (Chapitre 6), abandonne de façon visible si le retard ne se résorbe pas, et peut être arrêté et relancé sans perdre de travail.

La même règle s'applique aux deux tables append-only. Une migration peut ajouter une colonne à `ledger_entries`, mais elle ne réécrit pas les entrées existantes pour remplir cette colonne avec des valeurs corrigées ; une correction est une nouvelle entrée, émise par le chemin de correction de `payments-service`. Une migration qui touche effectivement ces tables est signalée par l'étape de classification ci-dessus et part en revue auprès de l'équipe plateforme.

DDL de hotfix et dérive

Chaque équipe finit par connaître la nuit où un index réglerait une requête lente, alors que le pipeline prend, disons, vingt minutes. La question est de savoir ce que le modèle d'accès dit du raccourci.

Il n'y a pas de raccourci par la console, alors construisez une lane rapide

Dans l'état cible, personne ne détient le login du propriétaire. `human_breakglass` écrit des `INSERT` et des `UPDATE` et ne porte aucun DDL, et le superutilisateur est un instrument scellé qui ne sert pas au travail quotidien (Chapitre 3). Une personne sur le primary n'a donc aucun moyen légitime d'ajouter un index à la main. C'est délibéré, et cela déplace la question vers le pipeline : à quelle vitesse peut-il exécuter un changement relu quand le besoin est urgent ?

Je donnerais au pipeline une **lane accélérée** plutôt qu'un contournement. La lane utilise le même job, le même credential et le même enregistrement ; elle diffère par le calendrier, pas par qui doit approuver. Le principe des quatre yeux demeure : un relecteur qui n'est pas l'auteur approuve toujours, et l'auteur ne fait pas le merge seul. Ce qui s'assouplit, c'est la période d'attente planifiée et, pour une courte allowlist de classes d'instructions, l'exigence de l'équipe plateforme, avec une revue de suivi par l'équipe plateforme le jour ouvré suivant. `classifyMigration` applique l'allowlist : construire un index, supprimer un index et ajouter une colonne nullable sont éligibles. Une instruction qui change des grants, la propriété ou une vue, touche une table append-only ou détruit des données ne l'est pas, parce que ce sont les changements pour lesquels la revue de l'équipe plateforme existe, et une migration qui en contient une est refusée dans la lane et passe par la lane normale. Le Chapitre 15 décrit l'accès pendant les incidents ; ce qui compte ici, c'est que l'urgence change la vitesse du processus, pas l'identité qui exécute le DDL ni la revue que reçoit une instruction dangereuse.

Le coût est que la lane accélérée est une option permanente pour sauter une part de la prudence, et une option de ce genre finit par servir. Suivez la fréquence de son utilisation, et traitez une hausse comme le signe que la lane normale est trop lente, pas comme celui que l'équipe est devenue moins prudente.

Dérive

La **dérive** est toute différence entre le schéma que décrivent les migrations et celui qui existe en production. Elle entre par un hotfix appliqué à la main, par une migration modifiée après avoir été appliquée, par un changement de niveau propriétaire fait par un outil dont personne ne s'est souvenu, et par des grants modifiés en dehors de la définition des accès. La dérive du schéma surprend la migration suivante. La dérive des grants est un changement d'accès qu'aucune revue n'a vu.

Une vérification planifiée peut trouver les deux types :

```
function detectDrift(environment):
    expected = buildExpectedSchema(appliedMigrations(environment))
    actual = readLiveSchema(environment)

    findings = []
    for each difference in compare(expected, actual):
        add(findings, difference)
    for each difference in compare(expectedGrants(),
readLiveGrants(environment)):
        add(findings, difference)

    if length(findings) > 0:
        notify("platform", "schema or grant drift in " + environment)
    recordCheckRun(environment, findings)

function checkDriftCheckAge(environment):
    // scheduled separately, so a stopped drift job cannot hide itself
    if now() - lastSuccessfulRun("detectDrift", environment) >
maxRunAge:
        notify("platform", "drift check has not completed recently")
```

Deux choses rendent cela utile. L'état attendu est dérivé des migrations et de la définition des accès du Chapitre 3 ; il y a donc exactement une source de vérité à laquelle comparer. Et la vérification couvre les grants, pas seulement les tables, parce qu'un grant qui apparaît en dehors de tout fichier est, pour ce livre, le type de dérive le plus important. Quand une différence est trouvée, la résolution est l'une des deux suivantes : écrire une migration qui enregistre le changement, s'il doit rester, ou en écrire une qui l'annule, s'il ne doit pas rester. Résoudre la dérive en modifiant l'enregistrement de ce qui a été appliqué cache le changement au lieu de le relire.

> **Remarque :** Ne « corrigez » pas la dérive en appliquant directement une instruction corrective en tant que superutilisateur ou propriétaire. L'instruction qui supprime la différence est elle-même un changement non relu en production. Mettez-la dans une migration, et faites-la passer par la lane.

Nouvelles tables, nouvelles colonnes, et les grants qui suivent

Le Chapitre 3 a choisi un défaut vide pour les nouvelles tables : une table n'est lisible par personne d'autre que le propriétaire tant qu'une migration n'a pas énoncé de grant. Cette section est l'autre moitié de cette décision, parce que c'est dans les migrations que la décision tient ou est oubliée.

Les grants vivent dans la migration

Chaque migration qui crée une table énonce ses grants dans le même fichier. Un relecteur voit alors tout l'effet d'un changement sur les accès dans un seul diff, ce qui est l'argument en faveur du défaut vide. Pour une nouvelle table, `invoice_adjustments` (une table illustrative, qui ne fait pas partie du schéma canonique), le fichier ressemble à ceci. Sa colonne de texte libre `reason` est classée Confidential par précaution pour le pire cas, comme l'est `invoice_lines.description` ; les lecteurs autres que le propriétaire de la table reçoivent donc des grants au niveau des colonnes qui l'excluent :

```
CREATE TABLE invoice_adjustments (
    id          text primary key,
    invoice_id  text not null,
    reason      text not null,
    delta_minor integer not null,
    created_at  timestamp not null
);

// the migration states every grant it means to give
GRANT SELECT, INSERT ON invoice_adjustments TO app_billing_api;
// reason is Confidential free text: left out of the broad read grants
GRANT SELECT (id, invoice_id, delta_minor, created_at) ON
invoice_adjustments TO app_reporting_job;
GRANT SELECT (id, invoice_id, delta_minor, created_at) ON
invoice_adjustments TO human_readonly;
```

Oublier un grant a deux issues possibles, et leur différence est la raison du défaut vide. Avec un défaut vide, le service qui a besoin de la table échoue à sa première requête. C'est bruyant, on le découvre lors d'un test ou au premier déploiement, et on le corrige par une migration d'une ligne. Avec un privilège par défaut large, un grant oublié est silencieux dans l'autre sens : la table suivante est lisible par quiconque le défaut désigne, et personne ne le remarque parce que rien n'échoue. Un échec bruyant coûte moins cher qu'une exposition silencieuse.

L'échec n'est bruyant que si quelque chose exécute la requête. C'est pourquoi j'inclurais dans le pipeline une étape de smoke test après la migration, qui vérifie que chaque rôle applicatif peut utiliser les tables qu'il est censé utiliser. Les rôles applicatifs n'ont pas de login ; l'étape a donc besoin d'un mécanisme. J'utiliserais un login de test membre de chaque rôle applicatif, qui passe à chacun tour à tour, là où le moteur le permet ; l'alternative est un login de test par rôle. Les deux choix élargissent la portée du pipeline, parce qu'un login capable de devenir n'importe quel rôle de service vaut plus pour un attaquant que n'importe quel credential de service isolé. Il est donc délivré par la workload identity du job, expire avec le job et ne sert qu'à cette étape. En échange, l'étape attrape le grant oublié avant que le déploiement du service ne le fasse.

Nouvelles colonnes sur des tables qui ont déjà des grants

Une nouvelle colonne se comporte différemment selon la manière dont sa table est accordée.

Style de grant de la table	Une nouvelle colonne est...	Conséquence
Grant au niveau de la table	lisible par chaque rôle qui détient le grant	Une nouvelle colonne sensible est exposée immédiatement
Grant au niveau des colonnes	lisible par personne tant qu'elle n'est pas nommée	Une nouvelle colonne doit être accordée pour être utilisée

Pour les tables qui ont des colonnes sensibles, comme `customers`, `bank_accounts` et `payment_tokens`, cela plaide pour des grants au niveau des colonnes là où le moteur les prend en charge, ce qui est le choix fait au Chapitre 3. La nouvelle colonne commence alors fermée, et la migration qui l'ajoute est l'endroit où l'équipe décide quels rôles peuvent la lire

et si elle est Confidential ou Restricted (le Chapitre 12 traite de la classification). Une migration qui ajoute une colonne à l'une de ces tables sans aucune instruction de grant appelle une question, et la vérification de classification peut la poser.

Là où le moteur n'a pas de grants au niveau des colonnes, la solution de repli du Chapitre 3 s'applique : les colonnes Restricted vivent dans des tables séparées comme `bank_account_secrets` et `payment_token_values`. Une nouvelle colonne Restricted a sa place dans l'une de ces tables, jamais dans la table principale, parce que les grants de la table principale sont au niveau de la table et que le plafond de l'accès JIT est tracé autour d'elle. Cette règle est facile à enfreindre par accident, puisque ajouter une colonne à la table que tout le monde connaît déjà est le geste naturel, et c'est le genre d'erreur que seul un relecteur qui connaît la solution de repli remarquera.

Les vues sur les tables sensibles

Sur de nombreux moteurs, une vue s'exécute avec les droits de son propriétaire, ce qui signifie qu'une vue sur une table avec des colonnes Restricted peut révéler ce que les rôles de ses lecteurs ne pourraient pas voir. C'est pourquoi `classifyMigration` signale `CREATE VIEW` comme un changement d'accès, pourquoi la migration qui en crée une passe par l'équipe plateforme, et pourquoi une vue sur une table sensible est jugée comme un changement d'accès et non comme une commodité.

Compromis et coûts

Le pipeline est une cible de grande valeur. Un credential capable d'exécuter du DDL de niveau propriétaire, détenu par un système que beaucoup de personnes peuvent modifier, est une concentration de pouvoir. Un pipeline relu n'empêche pas quelqu'un qui dispose d'une portée de niveau propriétaire de faire quelque chose de non relu ; il rend ce quelque chose visible après coup, ce qui est un contrôle plus faible qu'il n'y paraît. Les bornes de ce chapitre réduisent le risque sans le supprimer. Si la liste d'accès de votre pipeline est longue et que personne n'en est responsable, corrigez cela avant de vous appuyer sur le reste.

Les revues ajoutent du délai et ne valent que ce que vaut le relecteur. Un contrôle selon le principe des quatre yeux sur les migrations ralentit les petits changements et peut devenir une formalité, surtout quand le relecteur voit, disons, dix migrations par semaine et que chacune est une colonne. La vérification de classification aide en désignant les lignes dangereuses. Elle ne remplace pas l'attention. Une revue toujours approuvée en quelques secondes est le signal à surveiller, et le Chapitre 8 développe l'argument complet.

Expand and contract multiplie le nombre de changements. Un changement logique devient plusieurs déploiements. Pour une équipe avec une fenêtre de maintenance, un seul service et une petite table, cela n'en vaut peut-être pas la peine. Le pattern est rentable quand beaucoup de choses lisent la table et que personne ne peut les arrêter pendant la durée d'un changement.

Un lock wait court fait échouer les migrations. C'est le comportement voulu, et c'est un coût. Sur une table chargée, une migration peut demander plusieurs tentatives, et quelqu'un doit décider de ce qui se passe après la dernière. Si vous fixez une attente trop courte pour votre charge, le pipeline alertera l'équipe plateforme pour du bruit. Réglez-la d'après l'allure de vos propres longues transactions, et écrivez-le dans le runbook.

Les vérifications de dérive produisent des constats dont quelqu'un doit être responsable. Une vérification qui alerte un canal que personne ne lit ne trouve rien. Décidez qui est responsable de chaque classe de constat avant d'activer la vérification.

Quand ne pas utiliser cette conception. Pour une base de données jetable, un prototype ou un projet à un seul ingénieur sans données clients, un migrator avec un rôle propriétaire et un pipeline relu dépassent ce que la situation demande. Exécutez les migrations depuis la machine d'un développeur, gardez la base de production hors du chemin critique, et ajoutez le pipeline quand quelqu'un d'autre que vous dépend des données. La conception de ce chapitre vise le moment où le schéma est partagé par des services et où un changement erroné coûte quelque chose aux clients.

Erreurs courantes

- **Laisser le service créer ou modifier ses propres tables au démarrage.** Cela retire une étape de déploiement et donne au service la propriété, ce qui inclut le pouvoir de supprimer et de réattribuer des grants. Gardez la gestion automatique du schéma désactivée en production, et exécutez les migrations comme une étape séparée avec le migrator.
- **Garder le credential du migrator dans le gestionnaire de mots de passe d'une équipe « pour les urgences ».** C'est tentant parce que le pipeline échoue parfois au pire moment. Cela fait de chaque détenteur un propriétaire et supprime la garantie selon laquelle chaque usage des droits du propriétaire correspond à une exécution du pipeline. Construisez plutôt la lane accélérée.
- **Exécuter une migration sans lock wait.** La migration attend aussi longtemps que la plus ancienne transaction devant elle, et chaque instruction derrière elle attend aussi. Donnez au migrator un lock wait court et placez le retry dans le pipeline.
- **Cacher une correction de données dans un backfill.** Il semble efficace de corriger les valeurs fausses pendant que la table est de toute façon ouverte. La revue de la migration a été conçue pour une définition de colonne, pas pour une modification des montants des clients. Si le changement n'est pas un premier remplissage de la nouvelle structure selon une règle énoncée, c'est une correction de données, et elle suit le Chapitre 10.
- **Supprimer l'ancienne colonne dans la même version que la nouvelle.** Cela économise une migration et supprime le rollback. Gardez l'étape contract séparée, différée et relue comme destructive.
- **S'appuyer sur les privilèges par défaut pour que les nouvelles tables fonctionnent sans ligne de grant.** Un défaut large transforme un grant oublié en exposition silencieuse. Gardez le défaut vide et laissez chaque migration énoncer ses grants.
- **Ajouter une colonne Restricted à une table principale sur un moteur sans grants au niveau des colonnes.** Les grants de la table sont au niveau de la table, et le plafond JIT est construit autour d'eux. Placez les colonnes Restricted dans les tables de secrets séparées.

- **Traiter un constat de dérive comme du bruit.** Un grant qui existe en production et nulle part dans le code est une décision d'accès que personne n'a relue. Résolvez-le par une migration qui l'enregistre ou l'annule.
- **Modifier une migration après son application.** Les environnements qui ont exécuté l'ancienne version diffèrent désormais de ceux qui exécutent la nouvelle. Ajoutez plutôt une nouvelle migration.

Points clés

- Le DDL s'exécute en tant que `ledgerly_migrator` depuis le pipeline, en tant que propriétaire pendant la durée d'une migration, et aucun rôle applicatif ne détient de privilège DDL.
- Le migrator détient une autorité de propriétaire permanente, exercée au moyen d'un credential de courte durée : un seul job le détient, il est délivré par workload identity, chaque exécution est enregistrée avant de commencer, et les humains ne l'utilisent jamais.
- Comme le pipeline peut atteindre le pouvoir du propriétaire, sa propre liste d'accès et sa définition sont relues comme une migration ; les migrations qui touchent aux grants ou aux tables append-only vont à l'équipe plateforme, et la lane accélérée n'accepte qu'une allowlist de classes d'instructions que `classifyMigration` applique.
- Un lock wait court avec un retry dans le pipeline empêche une migration bloquée de fermer une table, et les outils en ligne et les étapes additives traitent les changements qui détiendraient sinon un verrou longtemps.
- Expand and contract permet à l'ancien et au nouveau code de fonctionner ensemble et fait du rollback un déploiement, au prix de davantage de migrations ; un backfill qui remplit la nouvelle structure selon une règle énoncée s'exécute dans la migration, et tout autre changement massif de données est une correction de données et suit le Chapitre 10.
- La dérive du schéma ou des grants se détecte en comparant l'état réel à ce que décrivent les migrations et la définition des accès, et elle se résout par une migration, pas par une modification discrète.

- Chaque migration énonce les grants de ce qu'elle crée et garde les défauts vides, si bien qu'un grant oublié échoue bruyamment au lieu d'exposer une table en silence.

Les changements de schéma sont le seul endroit où c'est un pipeline, et non une personne, qui détient le plus de pouvoir sur les données. La partie suivante passe de la question de savoir qui peut atteindre la base de données à celle de ce qu'elle contient : le Chapitre 12 demande quelles colonnes méritent une protection, comment les classer, et comment les masquer ou les retenir sans casser le travail qui en a besoin.

PARTIE IV

Protéger les données

Chapitre 12 — Données sensibles: classification, masquage et minimisation

Imaginez un lundi chez Ledgerly. Un client écrit au support pour signaler qu'un versement n'est pas arrivé et demande si le compte bancaire enregistré est le bon. Sam, au support, ouvre la page du client dans `admin-console` et trouve le numéro de compte complet à l'écran. Le relire dans le chat est le moyen le plus rapide de répondre, et une capture d'écran dans le ticket est le moyen le plus rapide de passer le dossier à un collègue. Personne dans cette histoire ne fait rien de déraisonnable, et en fin d'après-midi le numéro vit dans un log de chat et dans une pièce jointe de ticket.

Deux bureaux plus loin, Jordan, de l'équipe data, construit un rapport sur `ledgerly_analytics` et remarque que la table `customers` y contient des noms de contacts, des e-mails et des numéros de téléphone dont aucun rapport n'a besoin. Robin, d'astreinte, lit un log applicatif pour comprendre pourquoi une relance a échoué, et trouve l'adresse e-mail du destinataire dans le message d'erreur. Chaque exposition est petite, et les trois ont la même forme : des données sont arrivées à un endroit où personne n'avait décidé qu'elles devaient se trouver.

Ce chapitre ajoute une étape de l'état cible : les colonnes sensibles sont classifiées, masquées ou retenues par défaut, et la copie d'analytics ne contient aucune donnée Restricted brute. La question à laquelle il répond est ce que chaque personne et chaque copie ont réellement besoin de voir, et comment construire le système pour qu'il montre cela et rien de plus.

Dans le modèle en cinq parties de ce livre (**Identité, Grant, Chemin, Exécution, Enregistrement**), le chapitre travaille sur deux parties. Le Grant décide quelles colonnes un rôle peut lire tout court, et l'Exécution décide sous quelle forme une valeur revient par le chemin. Rien de cela ne dé-

pend de la manière dont un credential est parvenu à une personne ; un login parfaitement protégé qui lit un numéro de compte complet l'a tout de même exposé.

Le problème, et pourquoi il est difficile

Les contrôles des chapitres précédents décident qui atteint quelle table et quelles colonnes. Ils ne savent pas pourquoi une colonne compte. Un grant sur `customers` traite de la même façon un indicateur de statut et un e-mail de contact, et le schéma les déclare tous deux `text`. Rien dans la définition d'une colonne ne dit que `invoice_lines.description` est un texte libre qu'une personne a pu remplir avec un nom, ni que `iban` est le genre de valeur dont l'exposition ne peut pas être annulée en faisant tourner quelque chose.

La deuxième difficulté est que les gens ont besoin d'une partie des données, et qu'une règle qui dit que personne ne peut les voir produit des contournements. Le support doit répondre à des questions sur les comptes de versement. L'astreinte doit comprendre pourquoi une relance a échoué. L'équipe data doit compter les clients par segment. Quand le chemin autorisé ne montre rien, les chemins non autorisés apparaissent : captures d'écran, exports ad hoc, un grant permanent « jusqu'à la fin du projet ». La question utile n'est pas de savoir si une personne peut voir une valeur, mais de quelle forme de la valeur la tâche a besoin : la valeur entière, ses derniers chiffres, un oui ou un non, ou un décompte.

La troisième difficulté est que les données ne restent pas dans la table. Elles sont copiées dans un stockage d'analytics, écrites dans des lignes de log, reprises dans des messages d'erreur, collées dans des tickets et envoyées dans des exports. Chaque contrôle sur le primary s'arrête à la limite de la base de données, et la valeur poursuit son chemin. C'est pourquoi la réponse doit être construite dans le système et ne peut pas rester une politique. Une politique qui dit « traitez les données personnelles avec soin » ne peut pas être exécutée par un pipeline de migration, un chargement planifié ou un formateur de logs. Les décisions doivent être écrites dans les choses qui s'exécutent : le schéma, les grants, le chargement et le code de journalisation.

Les régimes de protection des données et des cartes de paiement, comme le RGPD, la KVKK ou PCI DSS, tendent à poser une famille de questions : quelles données personnelles détenez-vous, pourquoi, où vont-elles, qui peut les voir et pendant combien de temps. Une classification conservée dans le schéma est la forme sous laquelle vous pouvez y répondre. L'interprétation de tout régime appartient à vos collègues des services juridique et conformité ; ce chapitre décrit seulement l'ingénierie qui rend les réponses faciles à trouver. Le catalogue est une preuve de ce que fait le système, pas une affirmation qu'il satisfait un régime quelconque ; le Chapitre 14 revient sur les preuves qu'une revue des accès peut collecter.

Une classification qui vit dans le schéma

La **classification des données** range les données dans des classes nommées, et chaque classe détermine le traitement. Ledgerly en utilise quatre.

Classe	Ce qu'elle signifie	Exemples chez Ledgerly	Traitement par défaut
Restricted	Une exposition nuirait aux clients ou à l'entreprise d'une manière difficile à réparer	`iban`, `account_holder_name`, `token_value`	Retenue pour `human_readonly` ; jamais dans une demande just-in-time (JIT) ; modifiée uniquement par un chemin applicatif ; absente de la copie d'analytics
Confidential	Données personnelles ou commerciales sur les clients et les contacts	`contact_email`, `billing_address`, `provider_reference`, `invoice_lines.description`	Masquée ou retenue pour `human_readonly` ; démasquée seulement dans un `SELECT` JIT avec un approbateur ; masquée ou supprimée dans la copie d'analytics, sauf les indications d'affichage qui existent pour être montrés (`provider`, `expires_on` ; voir la table de décision plus bas)
Internal	Données opérationnelles sans contenu personnel	`amount_minor`, `status`, `due_on`, `account_code`	Lisible par les rôles qui en ont besoin ; copiée vers l'analytics

Classe	Ce qu'elle signifie	Exemples chez Ledgerly	Traitement par défaut
Public	Publiable sans risque	Aucune parmi les huit tables	Intégrité uniquement

La ligne Public est vide à dessein. Elle existe pour que « peut-on publier ceci ? » ait une autre réponse que le silence.

Quatre règles rendent les classes utilisables. C'est la partie que j'écrirais avant de choisir le moindre outil.

- **Classifiez les colonnes, pas les tables.** `customers` contient un indicateur de statut et un numéro de téléphone, et les deux demandent des traitements différents. Pour les seuls besoins de l'inventaire, une table prend la classe la plus élevée parmi ses colonnes.
- **Classifiez le texte libre pour le pire cas.** Une colonne dans laquelle des gens tapent contiendra à terme tout ce que des gens tapent. `invoice_lines.description` est Confidential parce qu'elle peut contenir des données personnelles, et aucun filtre ne peut dire quelles lignes en contiennent.
- **Une valeur dérivée hérite de la classe de sa source jusqu'à ce que quelqu'un argumente le contraire.** Un hachage à clé d'un `iban` est Restricted jusqu'à ce qu'une personne ait démontré qu'il ne l'est pas. `payment_tokens.last_four` est Confidential et non Restricted parce qu'il existe pour être affiché, et cette décision a été prise une fois et consignée par écrit.
- **L'inconnu est Confidential.** Une colonne que personne n'a classifiée est traitée comme Confidential et marquée provisoire, pour que le travail inachevé soit visible et que la valeur par défaut soit la valeur fermée.

Un catalogue dans le même dépôt que les migrations

Une classification qui vit dans un wiki dérive, parce que rien de ce qui modifie le schéma ne lit le wiki. Je la conserverais sous forme de fichier dans le dépôt qui contient les migrations, et je la modifierais dans la même pull

request que celle qui ajoute la colonne. Certains moteurs permettent aussi des étiquettes ou des commentaires sur les colonnes ; l'une ou l'autre forme convient tant que le pipeline peut la lire.

```
// illustrative catalog entry, one per column
table column_classification:
    table_name              text not null
    column_name             text not null
    class                   text not null    // Restricted,
Confidential, Internal, Public
    reason                  text not null    // one sentence, for the
reviewer
    owner                   text not null    // a team, never a person
    human_readonly          text not null    // readable, masked,
withheld
    analytics_copy          text not null    // copy, transform, drop
    reviewed_on             date null        // null means provisional
```

Le catalogue rend deux contrôles possibles. Le premier s'exécute sur chaque migration, à côté des signalements décrits au Chapitre 11. Contrairement à ces signalements, qui demandent à un relecteur de regarder, celui-ci bloque le merge, parce que classifier une colonne coûte une ligne et qu'une ligne manquante est précisément ce que tout le contrôle existe pour attraper.

```
function checkColumnClassification(migration, catalog):
    problems = []
    for each column in addedColumns(migration):
        entry = catalog.find(column.table, column.name)
        if entry is null:
            add(problems, "unclassified: " + column.table + "." +
column.name)
            continue
        if entry.class == "Restricted" and not
inRestrictedHome(column):
            // a main table whose grants are table-level cannot
withhold it
            add(problems, "Restricted outside its home: " +
column.name)
        if entry.class == "Restricted" and entry.analytics_copy !=
"drop":
            add(problems, "Restricted column would reach analytics")
    if problems is not empty:
        raise MergeBlocked(problems)
```

Ici, `inRestrictedHome` désigne une table où les grants peuvent retenir la colonne : une table principale avec des grants au niveau des colonnes, ou les tables de secrets séparées du Chapitre 3 sur les moteurs qui n'en disposent pas. Quand la personne qui ajoute la colonne a un doute, l'issue n'est pas de sauter le contrôle mais de classifier la colonne comme Confidential avec `reviewed_on` vide, et de laisser la prochaine revue des accès trancher.

Le second contrôle s'exécute de manière planifiée et compare le schéma réel au catalogue, de la même façon que le Chapitre 11 compare le schéma réel aux migrations. Une colonne créée à la main lors d'une mauvaise nuit apparaît comme un constat avec un responsable, et le constat se résout par une migration et une ligne de catalogue, pas par une modification discrète.

Minimisation: les données que vous ne détenez pas

La **minimisation** consiste à ne pas collecter, copier ni conserver ce dont personne n'a besoin. C'est le contrôle le moins cher de ce chapitre, parce qu'une valeur qui n'a jamais été stockée n'a pas de masque à maintenir, pas de copie à retrouver et pas de ligne à expurger. Tous les autres contrôles de ce chapitre supposent que les données existent ; celui-ci demande si elles devraient exister.

Pour chaque colonne sensible, je poserais quatre questions, dans cet ordre.

1. Une fonctionnalité ou une obligation a-t-elle besoin de cette valeur ? Si la réponse est « ça pourrait être utile », la réponse est non.
2. Pourrions-nous détenir un substitut à la place ? Ledgerly détient `payment_tokens`, des tokens émis par le fournisseur qui tiennent lieu d'instruments de paiement, et non les instruments eux-mêmes. Cette seule décision retire du schéma toute une classe de valeurs.
3. Pourrions-nous en détenir moins ? Une indication d'affichage comme `last_four` permet au support et aux clients de reconnaître un instrument sans l'instrument.
4. Pourrions-nous la détenir moins longtemps ? Les coordonnées d'un client clôturé ont une finalité jusqu'à ce que la dernière facture soit réglée et que les délais de contestation soient écoulés ; ensuite, elles ne

sont conservées que pendant une période courte et définie, puis supprimées.

Deux décisions sur l'endroit où vivent les données comptent davantage que n'importe quelle règle de rétention. La première est que **les tables append-only ne devraient pas contenir de données personnelles**. `ledger_entries` contient `amount_minor` et `account_code`, tous deux Internal, et c'est pourquoi une correction peut être une nouvelle écriture : rien dans la table n'a jamais besoin d'être effacé pour honorer une demande concernant une personne. `audit_events` désigne son sujet par un `subject_id` opaque pour la même raison. La seconde concerne le texte libre. Un champ structuré avec un ensemble fixe de valeurs peut être classifié et masqué ; une zone de notes ne peut être ni l'un ni l'autre, donc là où une fonctionnalité peut utiliser un code, un statut ou une référence produit au lieu d'une phrase, c'est ce que je choisirais.

La rétention est un job, pas une phrase dans une politique. Quelque chose doit s'exécuter de manière planifiée, trouver les lignes dont la période est dépassée et les supprimer ou les anonymiser, et ce quelque chose a besoin de ses propres privilèges restreints, de ses propres entrées d'audit et d'un responsable. Je ne connais pas la bonne période pour vos données, et ce livre non plus. Elle dépend de ce qu'exigent vos contrats, vos régulateurs et votre propre processus de contestation, et vos juristes doivent répondre à cette question avant qu'un ingénieur écrive le job. Ce qu'un ingénieur peut dire, c'est que les sauvegardes, les exports et la copie d'analytics contiennent eux aussi d'anciennes données, et qu'une suppression qui n'atteint que le primary n'est pas terminée. Le Chapitre 13 y revient.

Masquage: dynamique et statique

Le **masquage** remplace ou occulte une valeur pour que la vraie ne soit pas visible tout en gardant une forme utilisable. La **redaction** en est la version brutale : la valeur est retirée d'une sortie ou d'un log. Les deux types de masquage diffèrent par l'endroit où ils agissent.

	Masquage dynamique	Masquage statique
Ce qui change	Ce que renvoie une requête	La copie elle-même
Quand	À la lecture, par lecteur	Au chargement, une fois

	Masquage dynamique	Masquage statique
Valeur brute toujours stockée	Oui, dans le même stockage	Pas dans la copie
Usage typique	`human_readonly` sur une réplique	La copie d'analytics, les données de staging
Comment il échoue	Comportement du moteur, filtres, grants directs	Une copie périmée ou trop élaguée

Masquage dynamique pour les humains

`human_readonly` est la valeur par défaut avec laquelle chaque session commence, donc son masquage est celui que la plupart des gens rencontrent. La notation ci-dessous est illustrative, parce que les moteurs expriment le masquage de manières très différentes : certains proposent un masque sur la colonne, d'autres un privilège de démasquage, d'autres une vue fixe par table.

```
// illustrative notation. The grant lists the columns the role may
read;
// each mask rule changes what a read through the role returns
GRANT SELECT (id, created_at, status, contact_name, contact_email,
              contact_phone, billing_address) ON customers TO
human_readonly;

MASK RULE contact_name_initials   ON customers (contact_name)
    FOR human_readonly USING initials();
MASK RULE contact_email_partial   ON customers (contact_email)
    FOR human_readonly USING keep_first(1);
MASK RULE contact_phone_tail      ON customers (contact_phone)
    FOR human_readonly USING keep_last(2);
MASK RULE billing_address_hidden  ON customers (billing_address)
    FOR human_readonly USING redact();
MASK RULE provider_reference_tail ON payments (provider_reference)
    FOR human_readonly USING keep_last(4);

// free text has no shape to mask, so it is withheld: not in the
column list
// (illustrative column list)
GRANT SELECT (id, invoice_id, created_at) ON invoice_lines TO
human_readonly;
```

Le choix de la fonction découle de ce à quoi sert la colonne. Une technique n'est ni meilleure ni pire en soi ; chacune a sa place et sa manière d'échouer.

Technique	Ce que reçoit un lecteur	Convient à	Là où elle cesse d'aider
Partiel	Les quelques premiers ou derniers caractères	Un écran, un résultat collé	Les valeurs courtes se montrent presque entièrement
Redaction	Un marqueur fixe ou rien	Adresses, texte libre	Ne permet de confirmer rien
Pseudonyme à clé	Un substitut stable	Les jointures d'un chargement à l'autre	Faire tourner la clé casse les jointures
Généralisation	Une valeur plus large, comme un pays ou un mois	Analytics	Des colonnes combinées peuvent isoler une personne (Chapitre 13)

Un simple hachage ne figure pas dans la liste. Le hachage d'une valeur tirée d'un ensemble assez petit pour que chaque candidat puisse être essayé, comme un numéro de téléphone, peut être inversé en les essayant, donc un pseudonyme a besoin d'une clé détenue hors de la base de données.

Toutes les colonnes Confidential ne sont pas masquées. Une colonne qui existe pour être montrée reste lisible : `payment_tokens.provider`, `last_four` et `expires_on` sont les indications d'affichage qui permettent à une personne de reconnaître un instrument, et les masquer ne ferait que pousser les gens vers la valeur Restricted qu'ils remplacent. Les champs de contact, l'adresse de facturation et `provider_reference` identifient une personne ou une transaction et sont masqués. `invoice_lines.description` est retenue.

Cela donne la règle dont dépend le Chapitre 8. `human_readonly` lit déjà, donc un triplet `SELECT` JIT n'est nécessaire que pour une colonne Confidential que le rôle masque ou retient, et ce qu'il ajoute, c'est cette colonne démasquée. Les colonnes Restricted n'apparaissent jamais dans un triplet JIT, donc rien ne change ici pour elles. La demande et l'écran de l'approbateur devraient par conséquent porter une ligne explicite, « Colonnes démasquées : `contact_email` », distincte de la liste des tables, pour que l'approbateur voie quels masques la demande retire et pas seulement quelles tables elle touche.

> **Avertissement :** Vérifiez comment votre moteur traite un login qui lit une colonne via `human_readonly` et détient aussi un grant direct sur la colonne. Si le masque l'emporte, un `SELECT` JIT ne montre rien de nouveau et l'approbation ne signifie rien. Si les valeurs brutes l'emportent pour tous les lecteurs, le masque est décoratif. Testez les deux cas sur une copie avant de vous fier à l'un ou à l'autre.

Deux limites vont de pair avec la technique. D'abord, sur de nombreux moteurs, un masque s'applique à ce que renvoie une requête, pas à ce que comparent ses conditions, donc un lecteur autorisé à filtrer sur une colonne masquée peut apprendre quelque chose de ses valeurs par des décomptes et des correspondances. Considérez le masquage dynamique comme une protection contre l'exposition fortuite, comme un écran, une capture d'écran ou un résultat collé. Ensuite, masquer une sortie ne fait rien pour la valeur stockée, qui reste dans chaque sauvegarde, réplique et dump du primary.

La première limite donne une règle de décision pour l'inférence. Demandez-vous à quel coût un lecteur pourrait retrouver la valeur en filtrant : une poignée de sondages par égalité sur `contact_email` ne coûte presque rien, et un décompte des clients par préfixe de téléphone masqué non plus. Là où c'est bon marché et où le lecteur ne doit pas apprendre la valeur, retenez la colonne et ne la masquez pas, ou interdisez les prédicats sur la colonne masquée pour ce rôle. Là où c'est coûteux, comme sonder une longue valeur libre caractère par caractère à travers des milliers d'instructions journalisées, le masquage associé à l'enregistrement des instructions est un contrôle raisonnable.

Masquage dans le stream de résultats

Certains chemins d'accès ne peuvent pas compter sur le moteur pour masquer. Une gateway placée entre la personne et la base de données peut appliquer des règles au stream de résultats lui-même, avant que les lignes n'atteignent l'écran ou un fichier de résultats. Cela aide là où le moteur n'a pas de fonction de masquage, et les règles peuvent être les mêmes pour tous les moteurs situés derrière.

Une règle de stream dépend de la connaissance de la colonne d'où vient une valeur, et un alias comme `SELECT api_key AS k` met en échec les règles qui s'appuient sur le nom de la colonne. Les outils qui masquent les résultats, dont QueryProxy, ajoutent des règles qui s'appuient aussi sur la valeur, ce qui constitue une seconde couche, jamais un substitut au grant. Au moment où j'écris ces lignes, il documente lui-même cette limite des règles fondées sur le nom, et d'autres outils adoptent des approches similaires. La conséquence pour Ledgerly est qu'un masque de stream se place derrière les grants de colonnes du Chapitre 3, et qu'une colonne qu'un rôle ne doit jamais voir est retenue par le grant, et non laissée à un motif.

> **Remarque :** Sur les moteurs sans grants au niveau des colonnes, Ledgerly utilise la solution de repli du Chapitre 3 : les colonnes Restricted se trouvent dans `bank_account_secrets` et `payment_token_values`. Masquer `human_readonly` signifie alors généralement une vue masquée fixe par table, créée une fois par une migration et dont `ledgerly_owner` est propriétaire, le rôle détenant `SELECT` sur la vue et non sur la table de base. Un grant JIT au niveau de la table sur la table de base montre en clair toutes les colonnes non Restricted, Confidential comprises, et l'approbateur a accepté exactement cela. C'est une vue fixe, pas une vue créée à chaque demande, et c'est un mécanisme de masquage pour les colonnes Confidential. Ce n'est pas une seconde solution de repli pour les colonnes Restricted : la seule solution de repli pour celles-ci, ce sont les tables de secrets.

Masquage statique pour la copie d'analytics

La copie d'analytics est l'endroit où le masquage est le moins cher et le plus utile, parce que la valeur brute peut être entièrement laissée hors de la copie. Le rapport de Jordan a besoin de segments de clients, de décomptes et de montants. Il n'a pas besoin de noms, d'e-mails ni de numéros de téléphone, donc pour la plupart des colonnes Confidential le bon traitement est `drop`, et une transformation est réservée aux quelques colonnes dont les analystes se servent de façon démontrée.

Le chargement est un job planifié, pas un service, et il s'exécute sous son propre login et le rôle `app_analytics_load`, comme n'importe quel autre principal de la revue des accès. Le rôle détient `SELECT` uniquement sur les colonnes que le catalogue ne marque pas Restricted, donc le grant est le premier garde-fou et le catalogue le second. Ils ne sont pas indépendants, parce que le grant est dérivé des marques Restricted du catalogue : une colonne classée à tort comme Internal passe les deux. Ce que le grant ajoute, c'est qu'une colonne marquée Restricted ne peut pas être lue même si une entrée de transformation est erronée, et le contrôle indépendant est l'assertion de schéma dans `verifyMasked`, qui compare la copie au plan. Dans les limites de ce grant, le chargement décide colonne par colonne à partir du catalogue, et il échoue en mode fermé.

```
function loadAnalyticsTable(table, catalog):
    plan = []
    for each column in sourceColumns(table):
        entry = catalog.find(table, column)
        if entry is null:
            // an unclassified column never travels; the load says so
            notify("platform", "not loaded, unclassified: " + table +
"." + column)
            continue
        if entry.class == "Restricted" and entry.analytics_copy !=
"drop":
            raise CatalogInvalid
        if entry.analytics_copy != "drop":
            add(plan, { column: column, treatment:
entry.analytics_copy })
    // build beside the live table, check, then swap: a failure leaves
the
    // previous load in place, so the copy never holds a half-masked
table
    next = table + "_next"
    writeMasked(next, from: table, plan: plan)
    if not verifyMasked(next, plan):
        dropTable(next)
        raise MaskingCheckFailed
    swapTable(next, into: table)
```

`verifyMasked` effectue deux contrôles. Il vérifie le schéma : l'ensemble des colonnes de la cible doit être égal au plan, et aucune colonne Restricted ou supprimée ne doit y exister. Ensuite, il échantillonne des lignes et échoue si une colonne transformée est encore égale à sa valeur source. L'assertion de schéma attrape une colonne arrivée par un chemin que le

plan ne listait pas ; l'échantillon attrape une transformation mal configurée. Ni l'un ni l'autre ne prouve que la copie ne contient rien de sensible, parce que l'échantillon ne lit que certaines lignes.

Le prix de l'échec en mode fermé est visible : quand un développeur ajoute une colonne et oublie le catalogue, le contrôle de migration aurait dû l'arrêter, et s'il ne l'a pas fait, la table d'analytics se charge sans cette colonne et l'équipe data remarque un manque. J'accepterais cela. Un rapport avec une colonne manquante, c'est un ticket ; une copie avec une colonne non masquée, c'est une exposition que personne ne remarque.

Un pseudonyme à clé pour les jointures mérite une décision à part. Il permet aux analystes de suivre un même client d'un chargement à l'autre sans savoir qui est ce client, et il ne fonctionne que tant que la clé reste hors de portée de l'équipe data et inchangée. Une rotation de la clé change chaque pseudonyme, et l'historique ne se joint plus. Pour beaucoup d'équipes, le `customer_id` opaque fait déjà ce travail, et un second substitut ajoute une clé à protéger sans aucun gain.

Tokenisation et chiffrement au niveau de l'application

La **tokenisation** remplace une valeur par un substitut qui ne peut être reconverti que par un service contrôlé. Ce n'est pas du masquage, parce que la correspondance existe, et ce n'est généralement pas du chiffrement, parce que le substitut n'est normalement pas du tout calculé à partir de la valeur. Les schémas qui le calculent à partir de la valeur sous une clé sont du chiffrement sous un autre nom, et ils demandent le même soin pour la clé. Ledgerly en bénéficie déjà à un endroit : `token_value` est un token du fournisseur, le fournisseur détient la correspondance, et `payments-service` est le seul appelant. La base de données de Ledgerly ne détient donc jamais l'instrument.

Construire un second service de tokenisation pour les numéros de compte bancaire est un pas plus important. Cela ajoute au chemin de paiement un système qui doit être disponible quand les paiements le sont, il lui faut une liste des appelants autorisés à reconvertir un token, et chaque reconversion est un événement que quelqu'un devrait pouvoir examiner. Je ne le construirais pas à la taille de Ledgerly. Restreindre les colonnes, re-

streindre le seul service qui les lit et chiffrer les valeurs donnerait l'essentiel de la protection avec moins de choses à faire tourner. Si un partenaire régulé ou un prestataire de paiement l'exige, c'est l'exigence qui décide.

Le **chiffrement au niveau de l'application** signifie que le service chiffre une valeur avant que la base de données ne la voie, de sorte que les rôles, les répliques, les sauvegardes et les dumps contiennent tous du texte chiffré. C'est sa force distinctive : il protège la valeur dans les copies de données qu'un grant ne peut pas atteindre (Chapitre 13). Le chiffrement assuré par la couche de stockage, sur disque ou dans un snapshot, protège contre un volume perdu et ne fait rien contre une requête, et les équipes le confondent souvent avec un contrôle d'accès. Le chiffrement au niveau de l'application est plus proche d'une vraie frontière, et c'est un engagement plus lourd.

- **La clé décide qui peut lire.** Les données ne sont protégées qu'à la hauteur de leur clé. Donnez la permission de déchiffrement à la workload identity de `payments-service` et à rien d'autre, pour que la décision devienne une question de qui peut déchiffrer et par quel moyen. Les hiérarchies de clés, la rotation et la sauvegarde des clés sont une discipline à part entière, et la documentation du service de clés les couvre mieux que ce livre ne le peut.
- **La colonne perd ses fonctions de requête.** Vous ne pouvez pas rechercher, trier, filtrer par plage ni imposer l'unicité sur du texte chiffré de la manière habituelle, donc choisissez des colonnes qui ne sont jamais relues que par identifiant.
- **Un service qui peut déchiffrer à la demande peut se voir demander de le faire.** Le chiffrement déplace la frontière vers le service, donc ses appelants, son rythme d'appels et ses entrées d'audit deviennent le contrôle, et l'entrée d'audit est écrite avant le déchiffrement, comme ci-dessous.

```
function readPayoutAccount(accountId, onBehalfOf):
    row = selectBankAccount(accountId)       // the stored value is
ciphertext
    // the audit entry comes first: if the log is unreachable, nothing
is decrypted
    try:
        auditLogAppend({ actor: "payments-service", action: "decrypt",
                         subject: row.id, on_behalf_of: onBehalfOf })
    on AuditLogUnavailable:
        raise DecryptRefused
    // the key service checks the caller's workload identity, not the
request
    return keyServiceDecrypt(row.iban, keyId: row.key_id)
```

La matrice ne change pas si vous chiffrez. `human_breakglass` lit toujours les colonnes Restricted, mais ce qu'il lit est du texte chiffré, et le déchiffrement demande une permission sur la clé que le rôle ne porte pas. Je déciderais à l'avance si un incident qui a besoin du texte en clair peut demander un accès à la clé, avec quel approbateur, et je l'écrirais dans le runbook break-glass du Chapitre 8. Laisser ce choix pour l'incident, c'est ainsi qu'une règle finit improvisée.

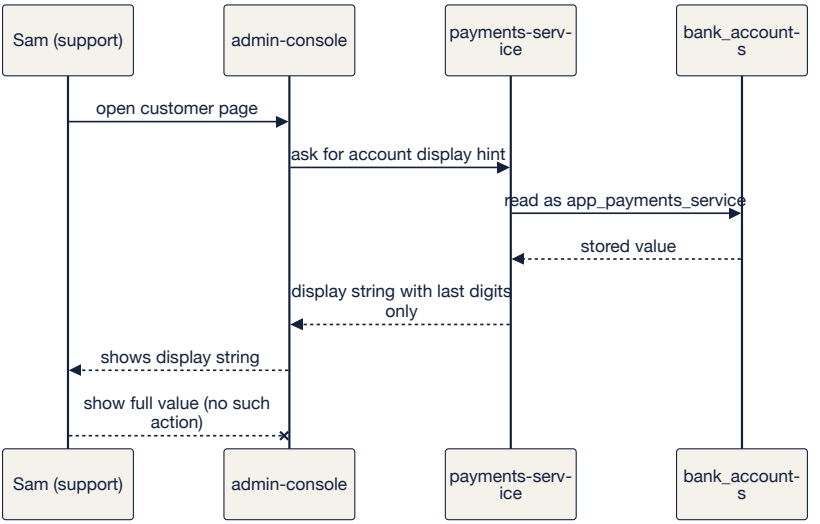
Travailler sans voir: le cas du support

Revenons à Sam. L'objectif est que Sam puisse répondre entièrement au client sans jamais voir la valeur complète. La plupart des questions du support s'avèrent avoir besoin de moins que la valeur.

Ce que le support doit faire	Ce qu'il a besoin de voir	Comment c'est fourni
Indiquer au client quel compte est enregistré	Les derniers chiffres	`payments-service` renvoie une chaîne d'affichage ; ou l'indice est stocké au moment où le compte est écrit, comme `last_four` l'est pour les tokens
Vérifier que la personne qui écrit est bien le contact	Un oui ou un non	Un contrôle restreint dans le service qui détient déjà le grant, qui compare ce que le client a dit à la valeur stockée et répond « correspond » ou « ne correspond pas »
Consulter les factures du client	Statuts, montants, échéances, pour ce client uniquement	`app_admin_console` lit les colonnes Internal de `invoices`, et la règle de ligne sur le client du dossier, issue du Chapitre 3, restreint les lignes

Ce que le support doit faire	Ce qu'il a besoin de voir	Comment c'est fourni
Corriger un numéro de compte erroné	Rien, et pas à la main	Le chemin applicatif : `admin-console` s'adresse à `payments-service`, une seconde personne approuve, et la modification est enregistrée dans `audit_events`

Le diagramme montre la première ligne. Regardez la dernière flèche : une action qui n'existe pas est un contrôle plus fort qu'une action qui existe et qui est refusée.



Je ne construirais pas de bouton « révéler » au départ. Un bouton de révélation qui demande un motif et journalise le clic semble raisonnable, et en moins d'un mois cliquer dessus est devenu une habitude et son log est quelque chose que personne ne lit. Si un vrai besoin apparaît, faites-en une demande avec un approbateur, comme le fait le Chapitre 8 pour toute lecture Confidential démasquée, et comptez combien de fois elle est utilisée. Une fonctionnalité utilisée tous les jours vous dit qu'il manque quelque chose à la forme d'affichage.

La personne d'astreinte suit la même règle sous une pression différente. Robin, qui débogue une relance échouée, a besoin de savoir qu'une adresse de destinataire a été rejetée, et a rarement besoin de l'adresse. Quand Robin a vraiment besoin de `contact_email` démasqué, c'est un `SELECT` JIT qui nomme la colonne, avec un approbateur qui lit la liste des

colonnes. Les lectures d'astreinte ne sont jamais approuvées automatiquement : une règle qui ignore les colonnes Confidential approuve une demande qui n'accorde rien, et une règle qui les inclut retire le seul contrôle qui a un sens. Le coût est une attente à une mauvaise heure, et la façon de le réduire est de supprimer le besoin de la valeur dans les cas courants, ce que couvre la section suivante.

Données sensibles dans les logs et les messages d'erreur

Les logs sont la copie de données que personne ne conçoit. Ils sont écrits par de nombreux composants, lus par de nombreuses personnes, conservés longtemps et fouillés avec des outils qui n'ont jamais fait partie du modèle d'accès. Quatre endroits portent l'essentiel des fuites.

- **Logs applicatifs et messages d'erreur.** Un message construit à partir d'une valeur, comme « impossible d'envoyer à l'adresse », met la valeur dans le log, dans l'outil de suivi des erreurs et parfois sur l'écran d'un client. Les drivers et les ORM peuvent aussi journaliser les paramètres liés quand le log de débogage est activé.
- **Logs de la base de données.** La journalisation des instructions peut enregistrer des littéraux, et un log des requêtes lentes conserve le texte d'une requête avec les valeurs qu'elle contient.
- **Enregistrements des instructions.** L'access gateway enregistre chaque instruction qu'une personne envoie (Chapitre 7), et une personne qui tape une valeur dans une condition a mis cette valeur dans l'enregistrement.
- **Tickets et chat.** Un résultat collé pour expliquer un problème est une copie avec le mauvais public et sans date de fin.

> **Avertissement :** Un log d'instructions contient des littéraux, donc le log d'audit du Chapitre 14 et les enregistrements d'instructions de la gateway sont eux-mêmes des stockages de données sensibles. Enregistrez les instructions sous forme de fingerprints avec les littéraux expurgés, classez le stockage au moins Confidential, restreignez qui le lit et donnez-lui une période de rétention, sinon le contrôle qui surveille les données devient une copie de plus de celles-ci. Le mécanisme relève du Chapitre 14.

Ma règle préférée pour la journalisation applicative est une allowlist, pas un scrubber. Un scrubber cherche des motifs qui ressemblent à des valeurs sensibles et les retire ; il attrape les formes auxquelles son auteur a pensé et manque le reste. Une allowlist nomme les champs qui peuvent apparaître et écarte tout le reste, de sorte qu'un nouveau champ reste silencieux jusqu'à ce que quelqu'un choisisse de le journaliser.

```
function buildLogEvent(event, allowedFields):
    safe = {}
    dropped = 0
    for each field in event.fields:
        if field.name in allowedFields:
            safe[field.name] = field.value
        else:
            dropped = dropped + 1
    // the count is the signal: a field dropped every day is either a
    // missing allowlist entry or a value somebody tried to log
    safe["dropped_field_count"] = dropped
    return safe
```

Les messages d'erreur suivent la même discipline. Une erreur porte un code et des identifiants, comme un identifiant de facture, un identifiant de client et un statut de livraison, et non la valeur qui a échoué. C'est aussi ainsi que le cas de relance de Robin cesse d'avoir besoin de l'adresse brute : si l'enregistrement de l'échec contient un statut comme « destinataire rejeté » et l'identifiant de la facture, le log répond à la question sans l'adresse, et l'enregistrement par la gateway de qui a lu l'adresse, si quelqu'un l'a fait, reste à un seul endroit.

La redaction dans le pipeline de logs est un filet de sécurité, et elle laissera passer des choses. Le contrôle que je ne sauterais pas est un canary dans les tests : exécutez l'application avec une valeur factice d'une forme recon-

naissable dans chaque champ sensible, et faites échouer le build si elle apparaît dans une quelconque sortie de log. Cela teste l'allowlist là où elle compte, et cela coûte quelques lignes. Le débogage devient plus lent quand le log contient des identifiants au lieu de valeurs, et c'est vers `dropped_field_count` que vont les messages « pourquoi je ne vois pas ça » qui en résultent, au lieu qu'un développeur réactive la journalisation des paramètres en production.

Les décisions de Ledgerly sur les colonnes

La table ci-dessous rassemble le chapitre. C'est l'enregistrement des décisions pour les colonnes sensibles des huit tables, sous la forme qu'une revue des accès peut lire. Dans cette colonne, « Retenue » et « Aucun accès » signifient tous deux que le rôle ne détient aucun grant sur la colonne, si bien qu'une requête qui la nomme est refusée. La colonne `human_readonly` indique la valeur par défaut pour une session ; la colonne JIT indique si une demande peut nommer la colonne.

Colonne	Classe	`human_readonly`	`SELECT` JIT	Copie d'analytics
`bank_accounts.iban`, `account_holder_name`	Restricted	Aucun accès	Jamais	Supprimée
`payment_tokens.token_value`	Restricted	Retenue	Jamais	Supprimée
`payment_tokens.provider`, `last_four`, `expires_on`	Confidential	Lisible	Pas nécessaire	`provider` et `expires_on` conservées, `last_four` supprimée
`customers.contact_name`, `contact_email`, `contact_phone`	Confidential	Masquée	Nomme la colonne, démasquée	Supprimée
`customers.billing_address`	Confidential	Expurgée	Nomme la colonne, démasquée	Supprimée

Colonne	Classe	`human_readonly`	`SELECT` JIT	Copie d'analytics
`payments.provider_reference`	Confidential	Quatre derniers caractères	Nomme la colonne, démasquée	Supprimée
`invoice_lines.description`	Confidential	Retenue	Nomme la colonne, démasquée	Supprimée
`amount_minor`, `currency`, `status`, `due_on`, `account_code`	Internal	Lisible là où la matrice l'accorde	Pas nécessaire	Copiée

Deux entrées méritent un commentaire. Les colonnes Restricted sont supprimées de la copie d'analytics et jamais proposées à un rôle humain autre que `human_breakglass` ; la seule route qui les modifie est le chemin applicatif, avec l'approbation d'une seconde personne (le Chapitre 9 explique comment la politique d'exécution y achemine les opérations soumises au gate). Le même chemin prend en charge le lien de propriétaire : `customer_id` sur `bank_accounts` et `payment_tokens` reste sous le plafond JIT parce qu'il n'est pas Restricted, mais déplacer une ligne vers un autre client change à qui appartient le compte, donc un changement de lien de propriétaire passe par le gate et est acheminé vers le chemin produit, jamais effectué comme une correction de données ordinaire. Et là où le moteur n'a pas de grants au niveau des colonnes, la même table reste valable, avec un changement : les colonnes Restricted vivent dans les tables de secrets, la colonne JIT indique « niveau table », et l'écran d'approbation le dit en toutes lettres.

Compromis et coûts

La classification demande de l'entretien. Un catalogue correct le jour où il est écrit commence aussitôt à vieillir. Quelqu'un doit être responsable de chaque décision de classe, revoir les classifications provisoires et répondre à la question que soulève une colonne non classifiée. Si personne ne revoit le catalogue, il devient le genre d'enregistrement complet et faux, ce qui est pire qu'un manque honnête.

Le masquage coûte du temps de débogage. Robin ne peut pas voir l'e-mail brut, donc un bug qui dépend de ses caractères exacts demande une demande JIT et un approbateur. J'accepte ce coût pour l'accès que ce chapitre ferme. Il n'est pas gratuit, et une équipe qui le paie chaque nuit commencera à chercher un contournement, donc mesurez la fréquence à laquelle la lecture démasquée est demandée et corrigez la cause.

Le masquage dynamique dépend du moteur. Le comportement vis-à-vis des filtres, des grants directs et des vues diffère, et un masque que vous n'avez pas testé sur votre moteur est un espoir. La variante à base de vues ajoute des objets que les migrations doivent maintenir alignés sur les tables de base.

Échouer en mode fermé produit des manques. Le chargement d'analytics qui refuse les colonnes non classifiées laisse l'équipe data sans une colonne jusqu'à ce que quelqu'un la classifie. Nommez qui est responsable de cette attente, et gardez la revue de classification assez rapide pour que le manque se compte en heures, pas en semaines.

Le chiffrement au niveau de l'application retire des fonctions et ajoute une dépendance. Le service de clés se trouve sur le chemin de chaque lecture de la colonne, et sa panne est une panne des paiements. Comme l'entrée d'audit vient en premier, un log d'audit injoignable arrête aussi ces lectures, ce qui est un couplage délibéré et une dépendance de disponibilité de plus. La perte des fonctions de requête pousse la recherche vers d'autres colonnes, et la sauvegarde des clés est une chose de plus qui doit être testée.

Certains contrôles de ce chapitre peuvent devenir du théâtre de sécurité. Un bouton de démasquage sur lequel on clique toujours, un catalogue que personne ne revoit, un scrubber de logs que personne ne teste et une colonne masquée que n'importe quel lecteur peut sonder avec des filtres ajoutent de la friction ou de la paperasse et ne réduisent rien.

Quand ne pas utiliser tout cela. Un petit système avec une poignée de tables et aucune donnée personnelle de client n'a pas besoin d'un fichier de catalogue, d'un chargement masqué ni d'un service de clés. Une liste d'une page des colonnes sensibles, des grants au niveau des colonnes qui les excluent et une allowlist de logs apportent l'essentiel de la valeur. Ajoutez le masquage quand un second public a besoin des données, ajoutez le

chiffrement quand une copie que vous ne pouvez pas contourner par des grants devient un souci, et notez par écrit quel événement vous ferait passer à l'étape suivante.

Erreurs courantes

- **Traiter le chiffrement au repos comme un contrôle d'accès.** C'est tentant parce que c'est un seul paramètre et que cela satisfait une checklist. Il protège un disque perdu et ne fait rien contre une requête. Gardez les grants, et utilisez le chiffrement au niveau de l'application là où une copie a besoin de protection.
- **Masquer dans le client et non dans la base de données.** Un masque dans l'interface web est facile à construire, et une connexion directe, un export ou une réponse d'API le contourne. Masquez là où les données quittent le stockage.
- **Classifier les tables et non les colonnes.** Une étiquette par table surprotège l'indicateur de statut et sous-protège le numéro de téléphone. Classifiez chaque colonne et dérivez-en l'étiquette de la table.
- **Mettre les nouvelles colonnes en Internal par défaut.** Internal est la classe de moindre friction, donc elle devient la classe de tout ce qui n'a pas été revu. Faites de Confidential la valeur par défaut et levez-la par une revue.
- **Masquer le texte libre avec des motifs.** Un motif trouve les numéros et les e-mails que son auteur attendait et manque un nom tapé dans une note. Retenez les colonnes de texte libre pour les lecteurs à large accès.
- **Utiliser un hachage sans clé comme pseudonyme.** Il paraît irréversible. Pour une valeur avec peu de possibilités, on peut l'inverser en les essayant. Utilisez un pseudonyme à clé dont la clé reste hors de la base de données, ou supprimez la colonne.
- **Copier la production vers le staging « et la masquer ensuite ».** La fenêtre entre la copie et le masquage est une copie de tout. Masquez au passage, comme le fait le chargement d'analytics, et traitez le staging comme l'une des portes du Chapitre 13.
- **Ajouter une colonne Restricted à une demande JIT pour gagner une étape.** Le plafond la refuse pour une raison, et la tentation est la

plus forte la nuit. Les valeurs Restricted ne changent que par le chemin applicatif avec une approbation selon le principe des quatre yeux, et sur les moteurs sans grants au niveau des colonnes elles se trouvent dans les tables de secrets.

- **Approuver automatiquement les lectures d'astreinte.** Cela retire la pause pendant laquelle une personne vérifie quelle colonne Confidential est démasquée. Gardez l'approbateur, et réduisez le nombre de fois où la valeur est nécessaire.
- **Journaliser les corps de requête au niveau error.** Le corps est la chose la plus utile pour un développeur et contient chaque champ qu'un client a tapé. Journalisez des identifiants et des statuts, et laissez une allowlist décider de ce qui apparaît en plus.

Points clés

- Classifiez chaque colonne, traitez le texte libre pour le pire cas, et faites en sorte que les colonnes non classifiées soient Confidential par défaut, pour que les nouvelles données commencent fermées et que le travail de leur ouverture soit visible.
- Conservez le catalogue de classification dans le dépôt qui contient les migrations, et bloquez un merge qui ajoute une colonne sans classe, parce que le contrôle coûte une ligne et que son absence est la défaillance.
- La minimisation vient en premier : des données qui n'ont jamais été stockées n'ont besoin d'aucun masque, et des données personnelles tenues hors des tables append-only ne peuvent pas devenir un problème d'effacement.
- Le masquage dynamique convient à `human_readonly` et protège contre l'exposition fortuite ; le masquage statique convient à la copie d'analytics, où la valeur brute peut être entièrement supprimée et où le chargement échoue en mode fermé sur tout ce qui n'est pas classifié.
- Donnez aux gens la forme d'une valeur dont leur tâche a besoin, comme une chaîne d'affichage ou un oui ou un non, et tenez les colonnes Restricted hors de toute demande JIT et de toute écriture en SQL brut.

- Un masque dans le stream de résultats est une seconde couche derrière le grant, jamais un substitut, et un rôle de chargement qui ne peut pas lire les colonnes Restricted fait échouer en mode fermé une erreur de catalogue.
- Les logs, les messages d'erreur et les enregistrements d'instructions sont des copies de données : journalisez des identifiants via une allowlist, tenez les valeurs hors des messages, et classifiez le log d'audit lui-même.

Les contrôles de ce chapitre protègent les données tant qu'elles se trouvent dans la base de données et tant qu'une requête les renvoie. Le chapitre suivant suit les données vers l'extérieur, dans les sauvegardes, les dumps, les exports et les copies de staging, où aucun de ces rôles ne s'applique.

Chapitre 13 — Sauvegardes, exports et autres portes latérales

Imaginez un vendredi après-midi chez Ledgerly. Robin traque un bug dans les relances de factures qui n'apparaît qu'avec des données réalistes : des factures aux échéances inhabituelles, des clients avec plusieurs contacts, des descriptions de lignes contenant du texte libre que personne n'avait prévu. Le staging compte une douzaine de clients créés à la main, et le bug n'apparaît sur aucun d'eux. Quelqu'un se souvient qu'un ancien dump de la production se trouve dans un stockage partagé, et Alex, qui gère les bases de données, le restaure dans le staging pour que toute l'équipe puisse regarder.

Le bug est trouvé dans l'heure. Le dump reste dans le dossier, et la base restaurée reste dans le staging. Le staging n'a pas d'access gateway, parce que ce n'est pas la production. Son login est partagé, chaque ingénieur peut l'atteindre, des jobs de CI s'exécutent dessus, et un prestataire arrivé le mois dernier y a un compte. `invoice-worker` dans le staging utilise les mêmes paramètres d'envoi d'e-mails que la production, et son job de relance est activé.

Personne n'a décidé d'exposer des données clients. Chaque étape était une réponse sensée au problème d'un après-midi, et les contrôles construits dans les chapitres précédents protègent une base de données qui ne détient plus la seule copie de son contenu.

Ce chapitre ajoute la dernière étape de l'état cible avant la Partie V : les sauvegardes, les exports et les copies de staging sont traités comme des portes avec des responsables et une date d'expiration. La question est la suivante : une fois que les données quittent le primary volontairement, pour la reprise, pour un fournisseur, pour un test ou pour une personne qui a besoin de regarder, comment garder les contrôles avec les données plutôt qu'avec la base de données d'où elles viennent ?

Dans le modèle en cinq parties (**Identité, Grant, Chemin, Exécution, Enregistrement**), une copie est un endroit où le Grant et le Chemin ne s'appliquent plus, parce que les données sont sorties par une route qui ne porte ni l'un ni l'autre. Le travail du chapitre consiste à faire porter un Grant à la copie elle-même : une liste de lecteurs, un responsable et une date de fin, avec un Enregistrement de qui l'a créée et de qui l'a supprimée.

Le problème, et pourquoi il est difficile

Le Chapitre 6 a défini une **porte latérale** comme une copie de données ou un chemin d'export qui contourne les contrôles d'accès du primary. Les répliques et la copie d'analytics sont des portes que vous construisez délibérément et que vous pouvez nommer. Ce chapitre traite du reste : sauvegardes et snapshots, dumps, fichiers CSV, bases de staging, quelques milliers de lignes sur un ordinateur portable, un cache de tableau de bord, un fichier dans la boîte de réception d'un fournisseur. La plupart ont été créées pour une bonne raison. Peu ont un responsable.

La difficulté centrale est que les contrôles d'accès vivent dans un système, pas dans les données. Le primary a une access gateway, un login par session, des grants au niveau des colonnes, des timeouts et un log d'audit. Un dump du primary n'a rien de tout cela, parce qu'un dump est un fichier. Quiconque peut lire le fichier a tout ce qu'il contient, y compris les colonnes que le schéma des rôles retient pour chaque rôle humain sauf `human_breakglass`. Une sauvegarde qui contient les colonnes Restricted est, pour les personnes qui peuvent la lire, un grant permanent sur ces colonnes qu'aucune gateway n'a émis et qu'aucune revue ne liste.

La réponse évidente est une politique : ne copiez pas les données de production. Elle échoue pour une raison simple. Copier, c'est ainsi que le travail se fait. La reprise a besoin d'une copie par définition. Un fournisseur a besoin de fichiers. Un bug qui dépend de formes de données réelles a besoin de quelque chose qui ressemble au réel. Une politique qui interdit tout cela est contournée, et les copies qui en résultent sont celles qui ne sont pas gérées. La meilleure question est de savoir quelles copies sont nécessaires, qui est responsable de chacune, qui peut la lire et quand elle prend fin.

Une deuxième difficulté est que ces portes sont créées par des équipes différentes pour des raisons différentes, si bien qu'aucun responsable ne les voit toutes. L'équipe plateforme est responsable des sauvegardes, l'équipe data de la copie d'analytics, l'équipe backend du staging, la finance de la relation avec le fournisseur, et chaque ingénieur du contenu de son propre ordinateur portable. Un inventaire est le premier contrôle, et il se trouve dans les buckets de stockage, les plannings de jobs et la configuration, pas dans un document de politique.

Une carte des portes latérales

Avant les contrôles, une carte. La table liste les portes que couvre ce chapitre, ce que chacune contient dans le cas de Ledgerly et qui l'atteint par défaut si personne n'a conçu autre chose.

Porte	Ce qu'elle contient	Qui l'atteint par défaut	Premier contrôle
Sauvegardes et snapshots de volume	Tout, y compris les colonnes Restricted et les tables de secrets	Quiconque a un accès au stockage ou à l'infrastructure	Pouvoirs séparés pour créer, restaurer et supprimer ; chiffrement avec sa propre clé
Dumps dans un stockage partagé	Tout, sous forme de fichier lisible	Quiconque peut ouvrir le dossier	Trouver, attribuer un responsable, supprimer ou placer sous les règles des sauvegardes
Exports humains (CSV, tables collées)	Tout ce que la requête a renvoyé	La personne, puis quiconque reçoit le fichier	Approbation par palier, expiration, aucune donnée Restricted
Bases de staging et de test	Une copie de la production, ou une imitation	Chaque ingénieur, la CI, les prestataires	Données masquées ou synthétiques, messages sortants désactivés
Ordinateurs portables	Jeux de résultats, bases locales, fichiers enregistrés	L'utilisateur du portable et ses sauvegardes	Supprimer le besoin ; échantillons masqués à la place
Outils de BI	Extractions en cache, rapports planifiés, tableaux de bord partagés	Toutes les personnes avec qui l'outil est partagé	Lire uniquement la copie d'analytics, identité par personne là où c'est possible
Fichiers fournisseurs	Exports de factures et du grand livre	Le personnel et les systèmes du fournisseur	Rôle restreint, liste de colonnes, rétention, expiration du fichier

Chaque ligne a la même forme de problème : une copie de données avec les contrôles d'un système de fichiers ou d'un produit SaaS, et une liste de lecteurs que personne n'a écrite. Le registre du Chapitre 6 existe déjà pour cette raison, et je l'étendrais à chaque ligne ci-dessus. Une porte qui ne figure pas dans le registre n'est pas une porte que quelqu'un examinera.

```
table data_copy:                       // extended from Chapter 6
    name         text primary key      // "staging_refresh_current"
    kind         text not null         // backup, dump, export,
staging, laptop,
                                       // bi_extract, vendor_file,
replica, analytics
    source       text not null         // what it was made from
    contains     text not null         // highest data class present
    readers      text not null         // roles, logins or groups
    storage      text not null         // where the bytes live
    owner        text not null         // a team, never a person
    ends         timestamp null        // null only for replicas and
analytics
    review_due   timestamp not null    // when the row is next reviewed
    deleted_by   text null             // who removed it, once removed
```

Les nouveaux champs sont `storage` et `deleted_by` ; `kind` gagne les valeurs `backup`, `laptop`, `bi_extract` et `vendor_file`, et `contains` enregistre la classe de données la plus élevée présente. Le premier nouveau champ dit où regarder quand quelqu'un demande ce qui peut lire la copie, parce que la réponse est généralement une permission de stockage et non un rôle de base de données. Le second boucle la boucle : une date `ends` sur laquelle personne n'agit est un vœu, et l'enregistrement de qui a supprimé la copie est la preuve que la date voulait dire quelque chose.

Sauvegardes et snapshots

Une sauvegarde est la seule porte latérale contre laquelle personne n'argumente, et cet accord explique en partie pourquoi elle est si souvent laissée sans contrôle. Elle doit tout contenir, parce que son rôle est de tout restaurer. C'est toute sa conception, et c'est aussi toute son exposition.

Ce que contient une sauvegarde

Les sauvegardes existent sous quelques formes. Une **sauvegarde physique** ou un snapshot de volume copie les fichiers de la base de données, souvent avec une archive continue des changements pour que la base

puisse être restaurée à un instant précis. Un **dump logique** lit les données à travers la base de données et les écrit sous forme d'instructions ou de lignes. Les deux contiennent toutes les tables. La forme physique contient aussi les définitions de rôles du cluster et tout le matériel de credentials que le moteur stocke pour eux, et certains outils logiques aussi, selon la façon dont ils sont utilisés.

Deux conséquences en découlent, et aucune n'est visible sur un diagramme du primary. La première est que les colonnes Restricted sont dans la sauvegarde, tout comme les tables de secrets sur les moteurs qui les utilisent (`bank_account_secrets` et `payment_token_values`, du Chapitre 3). La règle du schéma des rôles selon laquelle seuls les rôles applicatifs et `human_breakglass` atteignent ces valeurs décrit la base de données. Elle ne dit rien du fichier.

La seconde est que la sauvegarde contient l'état du jour où elle a été prise. Une sauvegarde antérieure au retrait du login admin partagé contient toujours ce login. Elle contient les appartenances aux rôles de personnes qui ont depuis changé d'équipe. La rotation et la révocation ont lieu dans le système vivant, et elles ne remontent pas dans les fichiers qui existent déjà. Quand quelqu'un restreint un rôle sur le primary, rien n'arrive à la restauration de l'an dernier : le primary s'améliore, et les copies gardent l'état du jour où elles ont été faites.

Restaurer, c'est lire

Je placerais une phrase en tête de la conception des sauvegardes : le pouvoir de restaurer est le pouvoir de tout lire. Quiconque peut restaurer une sauvegarde dans un endroit qu'il contrôle peut ensuite s'y connecter comme propriétaire et lire chaque colonne, et aucune gateway, aucun grant de colonne ni aucune règle de masquage de la production ne s'applique, parce que ce n'est pas en production que la lecture a lieu.

Cela change à qui appartient le contrôle. L'accès aux sauvegardes est généralement accordé par des rôles d'infrastructure : quelqu'un peut lire le bucket, monter le snapshot ou lancer l'outil de restauration. Ces rôles sont rarement revus avec le même soin que `human_readonly`, alors qu'ils se situent au-dessus. La question pour une revue des accès n'est donc pas seulement « qui peut interroger le primary » mais « qui peut restaurer, et vers où ».

Je séparerais trois pouvoirs, et je les donnerais à des principals différents :

Pouvoir	Ce qu'il signifie	Qui le détient
Créer	Créer des sauvegardes et les écrire dans le store	L'identité propre du job de sauvegarde, en écriture seule sur le store
Restaurer	Lire une sauvegarde et la remonter quelque part	L'équipe plateforme, par demande, avec l'approbation d'une seconde personne (principe des quatre yeux)
Supprimer	Retirer ou faire expirer des sauvegardes	Le processus de rétention ; les humains uniquement par une demande revue

L'intérêt de cette séparation est que le job qui écrit les sauvegardes ne peut pas les relire, et que personne ne peut à la fois restaurer une sauvegarde et supprimer les points de reprise, si bien qu'une restauration ne peut pas être suivie de la suppression de ce à partir de quoi elle a été faite. Là où le stockage prend en charge une rétention en écriture unique pendant une période, je l'utiliserais pour la même raison, avec un coût : vous ne pouvez pas supprimer une sauvegarde faite par erreur, comme une sauvegarde prise sur une base qui a brièvement contenu quelque chose qu'elle n'aurait pas dû, avant la fin de la période. Décidez à l'avance comment ce cas est traité.

Le chiffrement, et la clé de qui

Les sauvegardes devraient être chiffrées au repos, et la question intéressante est de savoir qui détient la clé. Si le même administrateur peut lire le bucket et utiliser la clé, le chiffrement protège contre un disque perdu et rien d'autre. Si la clé est détenue par un autre principal que les administrateurs du stockage, et que la restauration exige les deux, alors l'équipe de stockage seule ne peut pas lire les données et le détenteur de la clé seul n'a rien à lire.

Cette séparation a un coût précis. La restauration demande désormais deux parties, et une sauvegarde dont la clé est perdue n'est plus une sauvegarde. Un chemin de récupération de la clé doit exister et doit être testé, ce qui fait une procédure de plus à maintenir pour l'équipe plateforme. Le Chapitre 12 couvre le chiffrement au niveau de l'application et les limites

de la gestion des clés ; le point plus étroit ici est que le chiffrement des sauvegardes est un gate sur la restauration, et qu'un gate ne vaut la peine que s'il est différent de celui du stockage.

Le chiffrement n'aide pas contre la personne autorisée à restaurer. Il protège le fichier en transit et au repos contre les personnes et les systèmes qui ne font pas partie du chemin de restauration. Je le dirais clairement dans le document de conception, parce que « les sauvegardes sont chiffrées » est souvent entendu comme « les sauvegardes sont sûres », et ce sont deux affirmations différentes.

Une demande de restauration

La restauration est le moment où la sauvegarde devient une base de données vivante, donc c'est là que se place le contrôle. Le bloc ci-dessous montre une demande de restauration avec l'ordre des opérations écrit en toutes lettres. Le store des sauvegardes, le registre et le log d'audit sont des systèmes séparés, donc aucune transaction ne les couvre.

```
function restoreBackup(request):
    if request.approver is null or request.approver ==
request.requester:
        raise SelfApprovalNotAllowed
    if request.target not in isolatedTargets:
        raise TargetNotIsolated
    if request.ends is null or request.ends > now() + maxRestoreLife:
        raise RestoreNeedsEnd

    // the register comes first: a copy that exists with no record is
a side door
    registerCopy({ name: request.target, kind: "backup", source:
request.backup_id,
                   contains: "Restricted", readers: "none until access
applied",
                   owner: request.owner_team, ends: request.ends })

    try:
        auditLogAppend({ actor: request.approver, action: "restore",
                         subject: request.backup_id, target:
request.target })
    on AuditLogUnavailable:
        raise RestoreRefused

    // set before the copy exists, so a failure part-way still ends;
if this
    // call fails, the sweeper on `ends` removes the copy all the same
    scheduleDeletion(request.target, at: request.ends)
    // reachable only by the restore tooling until the last step
    startRestore(request.backup_id, into: request.target, reachable:
false)
    // every login returns with its backup-day credential, the
superuser and
    // ledgerly_access_admin included: nothing may connect before this
is done
    disableAllLogins(on: request.target)
    rotateSuperuserCredential(on: request.target)
    // the backup already holds a login with this name: drop it and
create it
    // again with a new credential, so the backup-day one never works
    replaceLogin("ledgerly_access_admin", on: request.target,
credential: "new")
    applyAccessDefinition(on: request.target)
    openTarget(request.target, readers: request.readers)
```

Si le processus s'arrête après `registerCopy` et avant le début de la restauration, le registre contient une entrée pour une copie qui n'existe pas, qu'un contrôle hebdomadaire trouve et retire. S'il s'arrête après le début de la restauration, la copie existe et est injoignable, et sa suppression planifiée la retire ; si même la planification a échoué, le sweeper qui agit sur `ends` dans le registre la trouve. L'ordre inverse laisserait une copie qui

existe sans être listée, et c'est cette défaillance-là qui compte. Le coût de cette conception est le délai : une approbation et une cible isolée avant que quiconque voie une ligne. Pendant un incident, ce délai est réel, et c'est pourquoi le chemin de restauration devrait être répété avec une approbation organisée à l'avance, comme le Chapitre 15 le présente pour les incidents. Quand le but est de récupérer quelques lignes et non une base entière, la restauration va vers la même cible isolée, les lignes y sont prises, et elles retournent en production comme une correction de données selon les règles du Chapitre 10.

Une copie restaurée démarre fermée

Une base restaurée ramène les rôles, les appartenances et le matériel de credentials du jour de la sauvegarde. Je la traiterais comme hostile jusqu'à ce que la définition des accès ait été appliquée de nouveau. En pratique, cela signifie une cible de restauration sans route réseau depuis les services et sans login humain, et une première étape qui désactive chaque login que la sauvegarde a ramené, le superuser et `ledgerly_access_admin` compris, parce que chacun revient avec le credential qu'il avait le jour de la sauvegarde. Ensuite, le credential du superuser est renouvelé, et seulement alors la définition des accès du Chapitre 3 est appliquée : les rôles, les appartenances et les grants tels que le code les décrit aujourd'hui. La copie devient joignable après cela, et pas avant.

Cette étape demande un administrateur des accès, et il doit appartenir à la cible. La sauvegarde contient déjà un login nommé `ledgerly_access_admin`, revenu avec le credential du jour où la sauvegarde a été prise. La restauration le remplace : le login ramené est supprimé puis recréé sous le même nom avec un nouveau credential, créé pour la restauration et supprimé avec elle. Le credential actuel de la production pour ce rôle n'est jamais copié dans la cible, et celui de la sauvegarde disparaît avant que quoi que ce soit puisse se connecter, parce que quiconque détient l'un ou l'autre peut accorder n'importe quoi à l'intérieur des rôles qu'il administre, et un exercice de restauration est exactement l'endroit où des credentials se retrouvent collés dans un environnement de test. Si l'administrateur propre de la copie restaurée est compromis, le rayon d'impact est cette copie.

Ce n'est qu'après cela que la copie restaurée reçoit des lecteurs, et les lecteurs sont les personnes indiquées dans la demande de restauration. Une restauration pour un exercice n'a besoin d'aucun lecteur humain : l'exercice vérifie que les données remontent et que les décomptes correspondent, puis la copie est supprimée.

Rétention et suppression

La rétention est un arbitrage entre deux coûts. Gardez les sauvegardes plus longtemps et vous pourrez vous remettre de problèmes remarqués tardivement, au prix d'une pile plus grande de données qui contient tout, y compris des données clients que le primary a supprimées depuis longtemps. Gardez-les moins longtemps et vous détenez moins, et une partie des corruptions sera plus ancienne que votre plus ancienne sauvegarde. Je ne connais pas la bonne période pour votre entreprise. Elle dépend du temps que met d'habitude un problème silencieux à être remarqué, de ce que coûterait un mois de données perdu, et de ce que vos collègues des services juridique et conformité disent des données supprimées qui survivent dans les sauvegardes. Les régimes de protection de la vie privée comme le RGPD ou la KVKK tendent à demander ce qu'il advient des données d'une personne une fois qu'elles sont supprimées du système vivant, et l'interprétation leur revient, pas à ce livre. Ce que l'ingénierie peut fournir, ce sont des preuves, pas la conformité : un registre et un enregistrement des suppressions qui montrent ce que fait le système.

Une conception qui mérite d'être discutée avec eux : les sauvegardes expirent selon un calendrier, et une procédure de restauration réapplique les suppressions enregistrées depuis la prise de la sauvegarde, pour qu'une copie restaurée ne ramène pas des données retirées volontairement. Cela ajoute une étape à chaque restauration et exige que les suppressions soient enregistrées dans un endroit qu'une restauration peut consulter, ce qui est une raison de plus de conserver les demandes de suppression dans un enregistrement durable. L'étape s'exécute après l'application de la définition des accès et avant l'ouverture de la copie.

```
function reapplyDeletions(target, backupTakenAt):
    // by execution time: a deletion recorded before the backup but
run after
    // it is not in the backup's state, so it must be applied again
    for each request in deletionRecord.executedSince(backupTakenAt):
        auditLogAppend({ actor: "restore-tooling", action:
"reapply_deletion",
                         subject: request.id, target: target })   //
record first
        deleteRows(on: target, subject: request.subject_id)
    // a failure here leaves the copy closed: openTarget has not run
yet
```

Les exercices créent des copies

Une sauvegarde qui n'a jamais été restaurée n'a pas prouvé qu'elle fonctionne. Les exercices de restauration sont la façon d'apprendre que les sauvegardes fonctionnent, et chaque exercice crée une base de données contenant des données Restricted. Je ferais passer les exercices par le même chemin de demande qu'une vraie restauration, avec la même cible isolée, la même entrée de registre et une date de fin dans la journée. L'exercice est aussi la répétition de l'incident : s'il a besoin d'un raccourci pour aboutir, ce raccourci sera pris à trois heures du matin.

> **Avertissement :** Une sauvegarde ou un dump qui contient des colonnes Restricted et se trouve dans un stockage lisible par un large groupe constitue la même exposition qu'un grant permanent sur ces colonnes, sans gateway et sans log. Si vous en trouvez un, restreignez d'abord la permission de stockage et parlez de rétention ensuite.

Dumps, exports et fichiers CSV

Les sauvegardes sont la porte qui a une finalité et une équipe. Les dumps et les exports sont les portes généralement ouvertes par une seule personne sous la pression du temps, et ils se présentent sous deux formes qui devraient être traitées différemment : les exports qui ont lieu selon un planning et ceux qu'une personne fait.

Les dumps faits à la main

Le dossier partagé de Ledgerly contient quelques anciens dumps. Chacun a été fait parce que quelqu'un avait besoin d'une copie pour une heure, et la copie est restée. Le dump d'une personne est une sauvegarde sans rétention, sans politique de chiffrement et sans responsable. C'est aussi, sur les moteurs où un dump lit à travers la base de données, la preuve que quelqu'un détenait un login capable de lire chaque table.

Je ferais trois choses. D'abord, sortir le dump de l'ensemble des outils normaux : l'access gateway ne propose pas de dump de la base entière, les rôles humains ne peuvent pas lire les tables dont un dump aurait besoin, et `human_breakglass` est le seul rôle humain qui le pourrait, avec l'ensemble de journalisation break-glass du Chapitre 8. Ensuite, trouver les dumps existants et, avant d'en supprimer un seul, vérifier si l'un d'eux n'est pas la seule sauvegarde de quelqu'un. Cela arrive : un dump devient la sauvegarde d'une table que personne d'autre n'a copiée. Construisez d'abord les vraies sauvegardes, puis supprimez les dumps. Enfin, donner à la personne qui avait besoin du dump une alternative qui fait le travail, ce qui est généralement la copie d'analytics masquée ou un jeu de données de test (voir la section sur les données de test plus bas).

Les exports humains

Les exports faits par des personnes sont les plus courants et les plus difficiles à interdire. Le support a besoin de la liste des factures d'un client. La finance a besoin d'un fichier des paiements du mois dernier. L'équipe data a besoin d'un échantillon. Je les classerais par palier selon la classe de données la plus élevée dans le fichier, parce qu'une règle unique qui convient à tous est soit trop lâche pour les dangereux, soit trop lourde pour les inoffensifs.

Palier	Contenu	Contrôle
Internal uniquement	Agrégats, décomptes, répartitions par statut, aucun contenu personnel	Libre-service, journalisé, sans approbation

Palier	Contenu	Contrôle
Contient du Confidentiel	Lignes avec des contacts clients ou du texte libre	Ticket avec un motif, un approbateur, une date de fin pour le fichier ; les lignes viennent d'une session `SELECT` JIT qui nomme les colonnes (Chapitre 8), si bien que l'approbateur lit la liste des colonnes que contiendra le fichier
Contient du Restricted	Valeurs de comptes bancaires ou de tokens	Pas exporté ; la réponse est une forme d'affichage ou un chemin applicatif (Chapitre 12)

Le palier est calculé à la gateway, pas déclaré par le demandeur, et il échoue en mode fermé : c'est la classe de catalogue la plus élevée (Chapitre 12) parmi les colonnes sources que la requête lit, et non parmi les noms des colonnes du résultat, parce qu'un alias ou une expression comme `lower(contact_email) AS x` masque la source. Une colonne dont la source ne peut pas être résolue, à travers une vue, une CTE ou une fonction, compte comme le palier exportable le plus élevé, et là où même cela ne peut pas être décidé, l'export est refusé. Le Chapitre 9 couvre les limites de l'analyse statique. Ces paliers illustrent une forme. Ce qui compte, c'est que l'approbation soit liée au risque. Une approbation accordée en quelques secondes, à chaque fois, pour tout, est le théâtre que décrit le Chapitre 8. Le palier qui demande un approbateur humain devrait être assez réduit pour que l'approbateur lise la demande, et je le mesurerais au fait que quelqu'un dise un jour non.

La date de fin est la partie que les équipes sautent. Un export sans date de fin est une copie qui survit à sa question. Le ticket qui approuve le fichier porte une date, le fichier va dans un endroit qui a un responsable, et un sweeper le retire. La règle du Chapitre 10 pour le travail ad hoc couvre le cas le plus léger : collez des identifiants et des décomptes, pas des lignes avec du contenu personnel. Le Chapitre 7 couvre ce que la gateway peut faire face à un gros jeu de résultats, à savoir le limiter et le rendre visible, pas l'empêcher.

L'export fournisseur

Le fournisseur comptable externe ne se connecte jamais à une base de données de Ledgerly. Il reçoit des fichiers, ce qui fait de l'étape d'export une porte que vous pouvez concevoir entièrement, parce que vous contrôlez les deux extrémités du chemin jusqu'à la boîte de réception du fournis-

seur. Dans l'état initial, l'étape d'export de `reporting-job` écrit ces fichiers avec le login admin partagé, si bien que le fichier peut contenir tout ce que le rôle admin peut lire. Dans l'état cible, elle utilise `app_vendor_export`, et cela change ce que le fichier peut contenir, parce qu'un fichier ne peut pas contenir plus que ce que le login qui l'a écrit pouvait lire.

```
// illustrative column lists: the vendor's real needs decide them
GRANT SELECT (id, customer_id, total_minor, currency, status, due_on,
created_at)
    ON invoices TO app_vendor_export;
GRANT SELECT (id, amount_minor, currency, account_code, created_at)
    ON ledger_entries TO app_vendor_export;
```

Le contrôle, c'est le grant, pas la requête. Une erreur dans la requête d'export peut omettre des lignes ou les répéter, et elle ne peut pas ajouter `contact_email`, parce que le rôle ne peut pas le lire. `customer_id` est un identifiant opaque et le fournisseur n'a aucune table à laquelle le joindre. Si le travail du fournisseur a besoin des noms des clients, c'est une décision que quelqu'un prend et approuve, et elle n'arrive pas parce qu'une requête a été élargie.

L'étape d'export se connecte à `ledgerly_replica_2`, comme le dit la politique de connexion du Chapitre 6, et écrit un fichier. Le fichier part ensuite vers le fournisseur par le canal convenu avec lui. Les étapes impliquent trois systèmes séparés, donc le bloc ci-dessous montre l'ordre et ce que laisse chaque défaillance.

```
function runVendorExport(period):
    file = writeExportFile(period, as: "app_vendor_export")  // local,
not yet shared
    manifest = { file: file.name, rows: file.rowCount, checksum:
file.checksum,
                 columns: file.columnNames, period: period }

    // the record exists before the file leaves: a sent file with no
    // record is the failure to avoid
    registerCopy({ name: file.name, kind: "vendor_file", source:
"ledgerly_replica_2",
                   contains: "Internal", readers: "accounting vendor",
                   owner: "platform", ends: now() + fileLife })

    try:
        sendToVendor(file, manifest)
    on SendFailed:
        // nothing has left; the file stays local and is removed with
its end date
        notify("platform", "vendor export not delivered: " +
file.name)
        raise ExportNotDelivered

    // the file has already left: a log outage delays the entry, never
the export
    auditOrQueue(manifest)
```

Ici, `auditOrQueue` ajoute l'enregistrement au log d'audit et, si le log est injoignable, le met en file d'attente pour un nouvel essai et prévient l'équipe plateforme, comme au Chapitre 7. Une défaillance du log devient un manque visible et réessayé, et non un manque silencieux, et aucune transaction ne couvre à la fois le log et le canal du fournisseur.

Le manifeste est la partie que les gens oublient. Il permet aux deux parties de vérifier que le fichier est bien celui qui était prévu, et il donne à une revue des accès quelque chose à lire : tant de lignes, ces colonnes, cette période. Si une colonne apparaît dans un manifeste alors qu'elle n'était pas dans celui du mois dernier, c'est un changement que quelqu'un aurait dû approuver.

Deux limites demeurent. Une fois que le fichier a atteint le fournisseur, sa rétention est une question de contrat et de confiance : vous pouvez demander au fournisseur de confirmer la suppression, et vous ne pouvez pas le vérifier. Et le checksum montre que le fichier n'a pas été altéré en transit, pas que le personnel du fournisseur est seul à le lire. Pour ces deux

points, je mettrais une clause de rétention dans le contrat et une ligne dans la revue des accès qui demande quand le fournisseur a confirmé la suppression pour la dernière fois.

Données de test: staging, copies masquées et données synthétiques

Le staging est là où le scénario de Ledgerly a commencé, et c'est la porte le plus souvent ouverte par gentillesse. Les ingénieurs veulent des données réalistes, et la production est la seule source de données qui a les formes que créent les vrais clients. Le coût est que le staging a le modèle d'accès d'un système de test, avec les données d'un système de production.

Trois types de données hors production

Type	Réalisme	Exposition	Coût
Copie de production	Le plus élevé	Identique à la production, avec des contrôles plus faibles	Peu coûteuse à créer, coûteuse à détenir
Copie masquée	Élevé, avec quelques formes perdues	Plus faible, pas nulle	Un pipeline de masquage à construire et à maintenir (Chapitre 12)
Données synthétiques	Plus faible, limité à ce qui a été modélisé	La plus faible	Un générateur à maintenir, et des bugs qui ont besoin de formes non modélisées

Je ne copierais pas de données de production non masquées dans le staging, et la raison n'est pas abstraite. Le staging a généralement plus de lecteurs que la production, un login partagé, une CI qui stocke des logs, et des intégrations configurées pour se comporter comme en production. Il combine le public le plus large avec les contrôles d'identité les plus faibles, et il détient les données les plus sensibles. Si les données doivent être réelles, les contrôles doivent être aussi forts que ceux de la production, et à ce stade le staging est la production sous un autre nom.

Ce qu'une copie masquée reste malgré tout

Le masquage, couvert au Chapitre 12, remplace ou occulte des valeurs tout en gardant une forme utilisable. Une copie masquée vaut bien mieux qu'une copie brute, et elle reste une copie de données, pour des raisons

qui méritent d'être énoncées afin que personne ne traite « masquée » comme un synonyme de « diffusable sans risque ».

Premièrement, le masquage couvre les colonnes que vous avez listées. Un texte libre comme `invoice_lines.description` peut contenir des données personnelles que la classification n'avait jamais anticipées, c'est pourquoi le livre le classifie pour le pire cas, et un masque qui l'ignore le laisse dans la copie. Deuxièmement, les masques qui préservent la forme gardent les lignes restantes distinguables : un montant de facture inhabituel, une date inhabituelle et un petit client peuvent ensemble désigner une seule entreprise. Les valeurs ont été masquées, et la combinaison porte encore de l'information. Troisièmement, le masquage cohérent, où une même entrée donne la même sortie pour que les jointures continuent de fonctionner, permet aussi de reconnaître la même personne d'une copie et d'une table à l'autre.

Rien de cela ne signifie que le masquage ne vaut pas la peine. Cela signifie qu'une copie masquée est traitée comme Confidential jusqu'à ce que quelqu'un ait revu ce qui reste, que son public est plus réduit que « toute l'ingénierie », et qu'elle a une date de fin comme toute autre copie ; la copie d'analytics continue est l'exception, et elle est revue à sa date `review_due` à la place. Le mot pour cela est **réidentification** : la capacité de déterminer de qui parle un enregistrement à partir des champs laissés en place. Je décrirais le risque à l'équipe en ces termes et non dans le langage des attaques. La réponse de conception est un public plus réduit et une expiration, pas une liste plus longue de champs masqués.

Données synthétiques

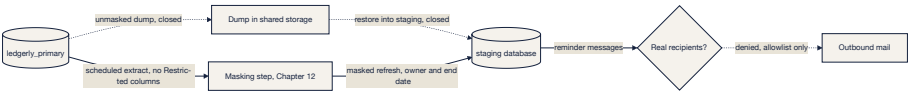
Les données synthétiques ne contiennent aucune personne réelle, donc elles portent l'exposition la plus faible. Leur limite est qu'elles ne contiennent que les formes que quelqu'un a pensé à modéliser, et les bugs qui ont besoin de données réelles sont ceux que personne n'a modélisés. Un générateur qui fabrique des clients bien rangés ne fabrique pas le client avec six contacts et une description en deux langues.

Je ferais grandir le jeu synthétique à partir des bugs. Chaque fois qu'un problème a besoin de données ayant la forme de la production pour être reproduit, la forme qui l'a causé, et non les valeurs, devient une fixture : un client avec plusieurs contacts, une ligne de facture avec une description

vide, une échéance en fin de mois. En un an, le jeu cesse d'être bien rangé et devient bizarre de la manière utile. Cela coûte une heure à un ingénieur après chacun de ces bugs, et cela supprime le besoin de la prochaine copie de production.

Le staging de Ledgerly, refait

Revenons au vendredi de Robin. Le chemin qui a ouvert la porte était un dump depuis un stockage partagé vers le staging. Le chemin que je construirais le remplace par un rafraîchissement qui passe par l'étape de masquage et aboutit dans une base de données à durée de vie courte avec un responsable nommé. Dans le diagramme, regardez les chemins en pointillés : ce sont ceux que la conception ferme.



La moitié droite de la conception est facile à oublier. Un environnement de staging détient des données et agit aussi sur elles. `invoice-worker` dans le staging lit les clients et envoie des relances, et si ses paramètres d'envoi correspondent à ceux de la production, il les envoie aux adresses de contact présentes dans les données. Une copie masquée n'a pas de vraies adresses, ce qui règle le problème par accident. Une copie brute transforme un test en messages envoyés à de vrais clients. Je ferais en sorte que le chemin sortant de chaque environnement hors production refuse par défaut et n'autorise qu'une allowlist d'adresses de test internes, quel que soit le contenu des données.

Les tokens de paiement suivent une règle voisine. Un token tient lieu d'instrument de paiement, et le fait qu'il fonctionne en dehors du système qui l'a émis dépend du fournisseur, ce dont je ne peux pas juger pour le vôtre. La règle par défaut est que `payment_tokens.token_value`, `bank_accounts.iban` et `bank_accounts.account_holder_name` ne sont jamais présents dans une copie hors production, masqués ou non (les sauvegardes sont l'exception, parce qu'elles doivent tout contenir), et que le staging utilise les tokens de test du fournisseur.

> **Remarque :** Traitez le rafraîchissement du staging comme un pipeline avec un chemin d'échec, pas comme une opération ponctuelle. Si l'étape de masquage échoue, le rafraîchissement s'arrête et le staging garde son contenu précédent. Un rafraîchissement qui se rabat sur une source non masquée le temps d'une exécution recrée le dump du dossier partagé avec quelques étapes de plus.

Outils de BI, ordinateurs portables et fournisseurs

Trois autres portes méritent un traitement bref, parce que chacune fait sortir les données de la base de données par une route qui ressemble à une fonctionnalité.

Outils de BI. Un outil de tableaux de bord se connecte avec un seul login, et chaque personne qui a accès au tableau de bord voit ce que ce login peut lire. Cela inverse le modèle d'identité du Chapitre 3 : la base de données voit un seul principal, et, disons, une centaine de personnes voient ses données. Deux contrôles aident. Pointez l'outil vers `ledgerly_analytics` et non vers une réplique, pour qu'il lise des données masquées avec un rôle propre à l'outil, un rôle `app_` comme pour n'importe quel autre workload (`human_analytics_reader` est le rôle de l'équipe data pour les personnes), au lieu de rôles de production. Et, là où l'outil le permet, faites-lui transmettre l'identité de chaque personne à la base de données, pour que les règles de lignes et de colonnes s'appliquent par personne. L'outil conserve aussi ses propres extractions, caches et rapports planifiés envoyés par e-mail, qui sont des copies de données qui vont dans le registre. Un tableau de bord est une porte pratique, et une extraction en cache est la même porte sans les contrôles.

Ordinateurs portables. La gateway retire les credentials enregistrés et fixe des valeurs par défaut en lecture seule, comme le décrit le Chapitre 7, et elle ne peut pas contrôler ce qu'une personne fait des lignes arrivées à l'écran. Le contrôle pratique consiste à réduire les raisons de garder une copie locale : un échantillon masqué, une base de staging qui se comporte comme la production, et des fixtures pour les formes qui causent des bugs. Une équipe ne peut pas prouver qu'aucun ordinateur portable ne contient de données de production, et je ne prétendrais pas le faire. Une base

comme le chiffrement complet du disque et un moyen d'effacer à distance un appareil perdu réduit ce que coûte un portable perdu, et c'est une base, pas une réponse.

Autres fournisseurs. La comptabilité est celui que suit le livre, et d'autres reçoivent des données par effet de bord : un outil de suivi des erreurs qui capture un corps de requête, un support desk qui reçoit des captures d'écran collées, une plateforme de logs qui stocke des instructions. Chaque fournisseur qui détient une copie reçoit une ligne dans le registre avec ce qu'elle contient et la date de fin du contrat. Le Chapitre 12 traite de la manière de tenir les valeurs sensibles hors des logs et des erreurs, ce qui est le moyen le moins cher de les tenir hors du stockage d'un fournisseur.

Fermer les portes chez Ledgerly

Voyons comment Alex ferait passer Ledgerly de copies éparpillées à un registre avec des responsables. Comme pour le chemin de lecture au Chapitre 6, l'ordre compte, et je procéderais en six étapes.

1. **Inventorier les copies.** Listez les sauvegardes, les snapshots, les dumps, les exports, le staging, les extractions de BI et les fichiers fournisseurs, avec leur stockage, leurs lecteurs et un responsable. Attendez-vous à trouver des copies dont personne ne se souvient ; la première passe du registre est surtout une découverte.
2. **Restreindre d'abord le stockage.** Pour tout ce qui contient des colonnes Restricted, restreignez qui peut lire le stockage avant de décider quoi supprimer. C'est rapide et réversible, et cela ferme l'exposition la plus large pendant que le reste se règle.
3. **Construire les pouvoirs sur les sauvegardes.** Donnez au job de sauvegarde une identité en écriture seule, placez la restauration derrière une demande avec un second approbateur, tenez la clé de chiffrement à l'écart des administrateurs du stockage, et faites passer un exercice par le chemin de demande.
4. **Retirer les dumps.** Une fois que les vraies sauvegardes existent et qu'une restauration a fonctionné, supprimez les anciens dumps et enregistrez qui a supprimé chacun. La restauration nocturne non masquée qui alimente `ledgerly_analytics` prend fin quand le chargement masqué du Chapitre 12, sous `app_analytics_load`, s'exécute à

sa place ; jusque-là, c'est la plus grande porte ouverte, et elle reçoit une date de fin.

5. **Faire passer le staging à des données masquées ou synthétiques**, avec les messages sortants désactivés par défaut, et supprimer le login partagé du staging.
6. **Donner à l'export fournisseur son propre login.** Faites passer l'étape d'export à `app_vendor_export`, ajoutez le manifeste et fixez une durée de vie au fichier.

Ensuite, reliez le registre à quelque chose qui a lieu de toute façon. La revue des accès du Chapitre 14 demande quelles copies existent et qui peut lire chacune, créer une copie exige une ligne dans le registre, et un sweeper retire tout ce qui a dépassé son `ends` après un délai de grâce et un avis au responsable. Les demandes de restauration et les exports devraient aussi écrire dans le log d'audit, qui est le sujet du Chapitre 14. Les jobs de restauration et d'export suivent la même discipline d'exécution que le runner qui porte les écritures approuvées : ils s'exécutent sous leur propre identité, à partir d'une demande approuvée, avec l'enregistrement écrit en premier, sauf qu'un export du palier Internal garde la règle de libre-service de la table des paliers ci-dessus (le Chapitre 9 décrit cette discipline).

Compromis et coûts

Le premier coût est la vitesse pendant un incident. Une restauration qui demande un approbateur et une cible isolée est plus lente qu'une restauration faite par quiconque détient la permission sur le bucket, et le temps qu'elle ajoute retombe sur la personne d'astreinte au pire moment. Répétez le chemin et organisez l'approbation à l'avance (le Chapitre 15 couvre l'accès pendant les incidents), pour que le contrôle soit une procédure que les gens connaissent et non un obstacle qu'ils découvrent.

Le deuxième coût est la garde des clés. Séparer la clé des sauvegardes du stockage signifie qu'une clé perdue est une sauvegarde perdue, et que le chemin de récupération de la clé est une procédure de plus qui doit être écrite et testée. Pour une très petite équipe, la séparation peut coûter plus qu'elle ne rapporte ; un seul store chiffré, des permissions de stockage strictes et une récupération de clé documentée peuvent être un point de départ raisonnable.

Le troisième coût est une boucle de débogage plus lente. Quand les ingénieurs ne peuvent pas restaurer la production dans le staging, certains bugs mettent plus de temps à être reproduits, parce que la forme qui les déclenche n'est pas encore dans les données de test. L'habitude des fixtures rembourse cela avec le temps et constitue une vraie dépense au départ. Les copies masquées ajoutent un pipeline dont quelqu'un est responsable, et il cassera quand une colonne sera ajoutée, c'est pourquoi la migration qui ajoute une colonne est l'endroit où sa classe se décide (Chapitre 11).

Le quatrième coût est la rétention. Une rétention plus longue garde plus d'historique récupérable et garde plus de données sensibles à plus d'endroits ; une rétention plus courte inverse les deux. Il n'existe pas de valeur juste pour tout le monde, et une équipe qui n'en a pas choisi une a choisi « pour toujours ».

Enfin, le registre et son sweeper sont une charge. Un sweeper qui supprime une copie dont quelqu'un avait encore besoin est un incident à part entière, donc la suppression devrait venir après un avis et un délai de grâce, et un responsable devrait pouvoir prolonger une date de fin une fois, avec un motif. Un registre que personne n'ouvre ne protège rien, et le lien avec la revue des accès est ce qui le garde honnête.

Quand ne pas construire tout cela : une petite entreprise avec une seule base de données, une poignée d'ingénieurs et aucune donnée Confidential digne de ce nom n'a pas besoin d'une approbation des exports par palier ni d'une clé séparée. Elle a quand même besoin de l'inventaire, d'un stockage restreint pour les sauvegardes et d'une règle selon laquelle rien de la production n'arrive non masqué dans le staging. Faites ces trois choses, notez par écrit ce qui vous ferait aller plus loin, et relisez cette note quand l'équipe ou les données changent.

Erreurs courantes

- **Restreindre l'accès au primary et ignorer ses sauvegardes.** Les rôles, une gateway et un log d'audit sont en place, et le bucket des sauvegardes est lisible par tout le groupe infrastructure. C'est tentant parce que les sauvegardes relèvent « des opérations ». Le remède est de

traiter la restauration comme une lecture de tout, avec son propre approbateur et son propre log.

- **Croire que « chiffré » veut dire « sûr ».** Le chiffrement au repos protège le fichier des personnes extérieures au chemin de restauration, et la clé se trouve à côté du bucket, entre les mains du même administrateur. Tenez la clé à part et dites contre quoi elle ne protège pas.
- **Supprimer les anciens dumps avant que de vraies sauvegardes existent.** Le dump était la seule copie d'une table pour quelqu'un, et le supprimer a supprimé le seul chemin de retour. Construisez et testez d'abord les sauvegardes, puis retirez les dumps.
- **Restaurer dans un réseau où les anciens logins fonctionnent.** La copie restaurée ramène les rôles et les credentials de son jour, y compris ceux retirés depuis. Restaurez en isolement, réappliquez la définition des accès, et remplacez le login `ledgerly_access_admin` ramené par la sauvegarde par un nouveau au lieu de garder l'un ou l'autre des anciens credentials.
- **Traiter une copie masquée comme publique.** Les valeurs sont masquées et la copie est largement partagée, puis un texte libre ou une combinaison rare de champs désigne un client. Traitez-la comme Confidential, avec un public plus réduit et une date de fin, jusqu'à ce qu'elle soit revue.
- **Rafraîchir le staging depuis la production comme une exception ponctuelle.** L'étape de masquage a échoué, la release attendait et le dump était juste là. Faites échouer le rafraîchissement en mode fermé, pour que les anciennes données restent et que rien ne se rabatte sur une source brute.
- **Laisser un outil de BI se connecter avec un rôle de production.** Un seul login derrière, disons, une centaine de personnes qui consultent, avec un cache que personne n'a listé. Connectez-le à la copie d'analytics avec son propre rôle, et enregistrez ses extractions dans le registre.
- **Approuver chaque export en quelques secondes.** L'approbation existe et ne réduit rien. Réservez l'approbateur au palier qui mérite une décision, et observez si quelqu'un refuse parfois.
- **Donner un responsable à une copie, mais pas de fin.** Le registre est complet et rien n'en sort. Chaque ligne a besoin d'une date `ends` et d'un sweeper qui agit dessus, avec un avis préalable.

Points clés

- Une porte latérale emporte les données sans les contrôles du système d'où elles viennent, parce que les contrôles d'accès vivent dans la base de données et non dans le fichier.
- Le pouvoir de restaurer une sauvegarde est le pouvoir de lire tout ce qu'elle contient, donc l'accès aux sauvegardes mérite la même revue que le rôle humain le plus privilégié.
- Créer, restaurer et supprimer des sauvegardes devraient être des pouvoirs détenus par des principals différents, et la clé de chiffrement devrait être tenue à l'écart des administrateurs du stockage, avec un chemin de récupération testé.
- Une copie restaurée revient avec les rôles et les credentials du jour où elle a été prise, donc elle démarre isolée, reçoit de nouveau la définition des accès et obtient son propre login administrateur.
- Un export fournisseur est contrôlé par le grant de `app_vendor_export`, pas par la requête, si bien que le fichier ne peut pas contenir une colonne que le rôle ne peut pas lire, et un manifeste permet de vérifier le fichier.
- Les données de test devraient être masquées ou synthétiques, une copie masquée est toujours traitée comme Confidential jusqu'à sa revue, et le staging devrait refuser les messages sortants par défaut et échouer en mode fermé quand le masquage échoue.
- Chaque copie a besoin d'un responsable et d'une date de fin dans un registre revu avec les accès, parce qu'une porte dont personne n'est responsable est une porte que personne ne ferme.

Les copies, les exports et les restaurations ont désormais un responsable et une date de fin, mais une conception ne vaut que par les preuves qu'elle est suivie. Le chapitre suivant passe à l'audit : où vit le log d'audit, ce qu'on y met, et comment se déroulent concrètement les revues des accès que ce livre ne cesse de mentionner.

PARTIE V

Prouver et exploiter

Chapitre 14 — Audit et imputabilité

Imaginez Ledgerly un mardi matin. Un client a demandé qui, dans l'entreprise, a consulté son compte de versement le mois dernier, et le contact sécurité transmet la question à Alex. Cela ressemble à une simple recherche. Alex commence par la liste des sessions de l'access gateway, qui montre que Robin a tenu une session break-glass sur le primary pendant quarante minutes. Alex ouvre ensuite le log de la base de données, qui nomme un login de session mais pas la raison de la session. Alex cherche enfin dans `audit_events`, qui enregistre les actions métier effectuées par l'application et ne dit rien de ce qu'une personne a tapé.

Les trois sources concordent sur l'essentiel de l'histoire et divergent sur le reste. L'horloge de la gateway a deux minutes d'avance sur celle de la base de données. Un login de session apparaît dans le log de la base de données sans aucune entrée dans la gateway, et personne ne peut dire s'il s'agit d'un login de service que la liste a oublié ou d'une connexion arrivée par un chemin que personne n'a conçu. Alex peut dire ce qui s'est probablement passé, mais ne peut pas le montrer, ni montrer que personne disposant des droits d'un administrateur de base de données n'a modifié la preuve.

Ce chapitre sert le pilier de l'imputabilité de la thèse, et il ajoute la dernière étape de l'état cible avant les incidents : l'accès est journalisé dans un stockage que le système accédé ne peut pas réécrire, et revu selon un calendrier. Dans le modèle en cinq parties (**Identité, Grant, Chemin, Exécution, Enregistrement**), c'est le chapitre de l'Enregistrement. L'Identité dit qui a agi ; l'Enregistrement est ce qui reste, et la question de savoir s'il relie l'instruction à la personne, à la demande et au résultat. La question ici est la suivante : que doit laisser derrière elle une instruction, où cela doit-il se trouver, et comment découvrir que la preuve est incomplète avant que quelqu'un ne pose la question ?

Le problème, et pourquoi il est difficile

Dans l'état initial, la question n'a pas de réponse. Le log de la base de données nomme un rôle partagé et aucune personne, et il se trouve sur des hôtes que les administrateurs contrôlent. Les chapitres précédents ont réglé la moitié qui concerne l'identité : un login par session, une demande par grant, un runner pour les écritures approuvées. Chacun d'eux ajoutait à l'enregistrement par le même appel, `auditLogAppend`, sans dire où vont les entrées. Ce chapitre répond à cette question.

La difficulté est que Ledgerly possède déjà trois enregistrements, et que chacun sait quelque chose de différent. Aucun d'eux n'est le log d'audit tel que le livre le définit : l'enregistrement de qui a fait quoi à quelles données, conservé là où l'acteur ne peut pas le modifier.

Enregistrement	Ce qu'il sait	Ce qu'il ne peut pas vous dire	Qui peut le modifier
Enregistrement des sessions de la gateway	La personne, la demande, la raison, l'instruction telle que soumise	Ce que la base de données a réellement fait ; s'il existait un autre chemin	L'équipe plateforme, via le stockage propre de la gateway
Log de la base de données	Le login, l'heure, le texte de l'instruction s'il est journalisé, les erreurs	La personne derrière le login, la demande, la raison	Quiconque administre l'hôte de la base de données
`audit_events`	L'action métier, le principal qui agit, le sujet	Les instructions qui l'ont produite ; tout ce que l'application n'a pas écrit	Personne par `UPDATE`, mais `ledgerly_owner` peut modifier la table dans une migration

La réponse évidente consiste à augmenter la journalisation : journaliser chaque instruction dans la base de données et la conserver longtemps. Cela ajoute un quatrième enregistrement et ne comble aucune des trois lacunes. Un log que le système accédé écrit sur lui-même n'est digne de confiance que dans la mesure où ses administrateurs le sont, et un log que personne ne lit relève du théâtre. Un log d'instructions est lui-même une copie de données sensibles, parce que les gens tapent des littéraux dans les conditions ; le Chapitre 12 en avertissait, et c'est ce chapitre qui porte le mécanisme.

L'enregistrement existe pour trouver la faille dans le système, non pour classer les personnes qui l'ont traversé. Quand une alerte se déclenche sur la session de Robin, la première question est ce que la conception a rendu nécessaire, et non ce que Robin a mal fait.

Une instruction, un enregistrement

Un enregistrement n'est utile que s'il peut répondre à une liste de questions écrites à l'avance. J'écrirais d'abord la liste et j'en déduirais les champs, parce qu'un stockage d'audit conçu à partir des lignes de log que la gateway se trouve produire répond aux questions que ces lignes de log se trouvent permettre.

La gateway écrit deux entrées par instruction, parce que le Chapitre 9 exige que l'intention soit écrite avant l'exécution de l'instruction et le résultat après. Ensemble, elles forment l'**enregistrement d'instruction** ; les deux entrées partagent un `statement_id`, et la seconde remplit les champs qui étaient inconnus au départ. Une seule forme sert aux deux, ainsi qu'aux enregistrements qu'écrivent d'autres principals.

```
table access_audit_record:
    stream          text not null      // one per producer instance
    sequence        integer not null   // gapless within a stream
    previous_hash   text not null      // hash of the previous entry
    entry_hash      text not null
    kind            text not null      // "intent", "outcome", "event"
    action          text null          // events only: grant, revoke,
breakglass, data-fix, contain, and so on
    statement_id    text null
    session_id      text null
    login           text not null      // what the database log shows
    principal       text not null      // the person or workload
identity
    on_behalf_of    text null          // the human behind an
automation principal
    roles           text not null      // role set at the time of the
statement
    request_id      text null          // always access_request.id; a
data fix has one too, and its fix_id resolves through it
    reason          text null
    approver        text null
    statement_class text null          // the Chapter 9 class the
classifier assigned
    fingerprint     text null          // statement shape, literals
redacted
    literals_hash   text null          // keyed hash of the literal
values
    tables          text null          // tables touched
    columns         text null          // columns touched, by data
class
    rows_affected   integer null       // outcome only
    rows_returned   integer null       // outcome only
    outcome         text null          // ok, refused, error, rolled-
back, row-cap-closed
    started_at      timestamp not null
    ended_at        timestamp null
```

Le type `event` porte tout ce qui n'est pas une instruction : une ouverture de session, un grant, une révocation, une rotation, une restauration, un début de migration. Ce sont les appels `auditLogAppend` et `auditOrQueue` des chapitres 3, 4, 7, 8, 10, 11, 12, 13 et 15, et la colonne `action` distingue un grant d'une révocation ou d'un confinement. Ils utilisent la même chaîne et les mêmes champs d'identité, si bien qu'une seule requête peut suivre une personne d'une demande à un grant, puis aux instructions qui l'ont utilisé, puis à la révocation.

La table ne vaut la peine d'être construite que si elle répond aux questions que les gens posent. La seconde liste est celle que je collerais à côté du schéma.

Question	Champs qui y répondent
Qui a exécuté cette instruction ?	`principal`, `on_behalf_of`, `login`
Avec quels grants et quelle approbation ?	`roles`, `request_id`, `approver`, `reason`
Qu'a-t-elle demandé ?	`statement_class`, `fingerprint`, `literals_hash`, `tables`, `columns`
Qu'a-t-elle fait ?	`outcome`, `rows_affected`, `rows_returned`
A-t-elle touché des données Restricted ?	`columns` classées par classe de données, avec `rows_returned`
Quand, et pendant combien de temps ?	`started_at`, `ended_at`
Manque-t-il quelque chose autour ?	`stream`, `sequence`, `previous_hash`, l'intention sans résultat
Cette entrée est-elle telle qu'elle a été écrite ?	`entry_hash`, vérifié par `verifyChain` ci-dessous

Deux champs portent l'essentiel du poids. `principal` et `on_behalf_of` empêchent la piste de s'arrêter à une machine, comme le Chapitre 9 l'exige pour l'automatisation. Le `request_id` est la clé de jointure vers tout ce que les chapitres 8 et 10 ont enregistré.

Un enregistrement sans `request_id` reste valide. Une session `human_readonly` de routine n'a pas de demande, et son enregistrement le dit en laissant le champ vide. Un `request_id` vide sur une session qui détient des triplets `human_jit_write` ou `human_breakglass` est un constat, et les contrôles décrits plus loin dans ce chapitre le recherchent.

Un stockage que le système accédé ne peut pas réécrire

L'endroit où vivent les entrées compte davantage que leur format. Le Chapitre 1 a défini l'observabilité comme un enregistrement qui se trouve quelque part où le principal ne peut pas le modifier. Les principals qui comptent ici sont ceux qui ont le plus de pouvoir : les administrateurs de base de données, l'équipe plateforme qui exploite la gateway, et `ledgerly_access_admin`, dont les trois logins peuvent accorder presque

n'importe quoi au sein des rôles qu'il administre. Si l'un d'eux peut modifier l'enregistrement, l'enregistrement ne prouve que ce qu'ils ont choisi de laisser.

Je construirais le stockage avec cinq propriétés, dans cet ordre.

1. **Un compte séparé.** Le stockage d'audit vit dans un compte cloud, un projet ou un groupe d'hôtes dans lequel les administrateurs de base de données, les opérateurs de la gateway et les trois logins `ledgerly_access_admin` n'ont aucun droit. Quelqu'un doit l'administrer ; c'est un petit groupe qui ne détient aucun droit d'administrateur de base de données, et son propre usage du stockage est visible par quelqu'un d'extérieur à ce groupe.
2. **Un producteur en écriture seule.** Chaque producteur (la gateway, le runner, le credential broker, le pipeline, le job de révocation, l'outillage de restauration) détient son propre credential, qui peut appeler une seule opération : l'append. Il ne peut ni lire, ni mettre à jour, ni supprimer. Un credential de producteur qui a fuité peut ajouter de fausses entrées, ce qu'une chaîne rend détectable, et ne peut pas en retirer de vraies.
3. **Un service d'append qui détient la chaîne.** Les producteurs ne calculent pas de hachages. Un petit service, à l'intérieur du compte d'audit, charge le head du stream, chaîne la nouvelle entrée et stocke les deux. Un producteur qui calculerait son propre maillon pourrait aussi recalculer tout ce qui le suit.
4. **Une rétention que l'administrateur du compte ne peut pas raccourcir.** Là où le stockage propose un verrou de rétention à écriture unique, utilisez-le pour la durée de rétention, afin que supprimer une entrée avant son terme ne soit possible pour personne dans le compte. Là où il ne le propose pas, la déclaration honnête est que l'administrateur peut supprimer, et ce sont la chaîne et l'ancre décrite ci-dessous qui rendent une suppression visible.
5. **Une ancre.** Une fois par jour, le service d'append envoie le hachage du head de chaque stream à un second endroit, détenu par un groupe différent : le système des contacts sécurité, ou un second stockage. Sans ancre, une personne qui contrôle entièrement le stockage pourrait réécrire un stream à partir d'un certain point, en recalculant chaque hachage qui suit. Avec une ancre, le head réécrit diffère de l'ancre.

```
function appendChained(stream, record):
    // runs inside the audit account; producers can call only this
    // one writer per stream at a time: the head is read and locked
    // inside the same transaction that stores the next entry
    transaction:
        head = lockHead(stream)
        entry = { stream: stream, sequence: head.sequence + 1,
                  previous_hash: head.hash, body: record,
                  stored_at: now() }
        entry.entry_hash = hashOf(entry.stream, entry.sequence,
                                  entry.previous_hash, entry.body)
        // one system, so one transaction is allowed: the entry and
the
        // new head exist together or not at all
        storeEntry(entry)
        storeHead(stream, entry.sequence, entry.entry_hash)
    // commit point: the entry is durable and the chain has moved
    return { sequence: entry.sequence, hash: entry.entry_hash }
```

Deux sessions peuvent faire un append sur le même stream au même moment ; le head est donc lu sous verrou à l'intérieur de la transaction. Sans ce verrou, les deux liraient le même head, écriraient le même numéro de séquence et provoqueraient une bifurcation de la chaîne, et le vérificateur signalerait alors une altération qui n'a jamais eu lieu. La règle est un seul écrivain par stream à la fois, imposé par ce verrou.

Le vérificateur est un job séparé, doté d'un credential en lecture seule, dans le même compte. Son travail est la partie qui transforme un stockage en preuve :

```
function verifyChain(stream, from, to):
    previous = loadEntryBefore(stream, from)
    for each entry in entries(stream, from, to):
        if entry.sequence != previous.sequence + 1:
            notify("security", "hole in " + stream + " at " +
entry.sequence)
        if entry.previous_hash != previous.entry_hash:
            notify("security", "broken link in " + stream)
        if entry.entry_hash != hashOf(entry.stream, entry.sequence,
                                      entry.previous_hash, entry.body):
            notify("security", "entry altered in " + stream)
        previous = entry
    // compare at the anchor's own position, not at the end of the
range
    anchor = loadAnchor(stream, atOrBefore: to)
    if anchor is not null and anchor.hash != entryAt(stream,
anchor.sequence).entry_hash:
        notify("security", "head differs from anchor in " + stream)
```

Vient maintenant la limite honnête, que je placerais dans le document de conception comme une phrase à part entière. Cette conception fournit une preuve, pas une prévention. Elle n'empêche pas une instruction de s'exécuter, et elle n'empêche pas quelqu'un qui a suffisamment de pouvoir sur le compte d'audit de détruire le stockage. Elle rend l'altération détectable et réduit les personnes qui pourraient la commettre à une liste courte et séparée. La détection dépend aussi de quelqu'un qui lit la sortie de `verifyChain` ; le vérificateur est donc un job planifié doté de son propre heartbeat, traité dans la section suivante.

Le coût est réel. Le compte d'audit est un système de plus à maintenir disponible, et les chapitres 8 et 9 en ont déjà accepté la pire conséquence : un grant, une déclaration break-glass ou une instruction est refusé quand son intention ne peut pas être écrite. Une panne du stockage d'audit bloque donc le travail privilégié, et elle le bloque au moment où quelqu'un attend peut-être une urgence. C'est le prix d'un enregistrement qui ne dépend pas de la bonne volonté.

> **Avertissement :** Ne placez pas le stockage d'audit dans la base de données qu'il décrit, même dans un schéma qui a son propre propriétaire. Quiconque peut administrer cette base de données peut modifier les entrées, et un rôle limité à `INSERT` n'y change rien. La seule exception structurelle que le livre recense déjà, à savoir que `ledgerly_owner` peut modifier `ledger_entries` et `audit_events` dans une migration, est la raison pour laquelle `audit_events` est l'enregistrement métier de l'application et non le log d'audit.

Trouver les lacunes

Une chaîne prouve que les entrées présentes n'ont pas été altérées. Elle ne prouve pas que chaque instruction a produit une entrée. La seconde famille de contrôles cherche ce qui devrait exister et n'existe pas. Il y a quatre sortes de lacunes, et chacune a une signification différente ; le contrôle les signale donc séparément.

Trous de séquence. Un stream est sans trou par construction ; un trou signifie donc que le service d'append a perdu une entrée ou que quelqu'un en a retiré une. C'est le constat le plus rare et le plus grave.

Intentions sans résultat. La gateway écrit une intention avant une instruction et un résultat après. Une intention sans résultat au-delà de l'instruction la plus longue que le timeout autorise signifie que la gateway a échoué entre les deux, que le résultat attend dans la file de retry, ou que l'instruction s'exécute encore. Seul le troisième cas est inoffensif, et il ne devrait pas survivre au timeout.

Sessions d'un seul côté. Chaque login de base de données devrait avoir une origine connue. Pour les humains, l'origine est une session de la gateway. Pour les services, c'est un lease du credential broker du Chapitre 4. Pour les migrations, c'est un run du pipeline. Un login présent dans le log de la base de données sans aucune des trois est un chemin que personne n'a conçu, et une session de la gateway qui revendique des instructions que la base de données n'a jamais vues est un enregistrement qui décrit quelque chose qui n'a pas eu lieu.

Producteurs silencieux. Un stream qui cesse d'écrire peut signifier une semaine calme, un producteur mort ou un producteur que quelqu'un a désactivé. Le contrôle compare chaque stream à une cadence attendue.

```
function detectGaps(window):
    gatewaySessions = sessionsFromAuditStore(window)
    databaseLogins = loginsInDatabaseLog(window)
    for each login in databaseLogins:
        if login not in gatewaySessions
           and not hasBrokerLease(login, window)
           and not isPipelineRun(login, window):
            notify("security", "login with no known origin: " + login)
    for each s in gatewaySessions:
        if s.statements > 0 and s.login not in databaseLogins:
            notify("platform", "gateway session with no database
login: " + s.id)
    for each i in intentsWithoutOutcome(olderThan: maxStatementAge):
        if not inRetryQueue(i):
            notify("platform", "intent with no outcome: " +
i.statement_id)
    for each stream in expectedStreams():
        if now() - lastEntry(stream) > expectedCadence(stream):
            notify("platform", "stream silent: " + stream)
```

Le log de la base de données atteint le compte d'audit par un agent d'expédition doté de son propre credential en écriture seule. Quelqu'un qui a accès à l'hôte peut arrêter cet agent, et c'est à cela que sert le contrôle des

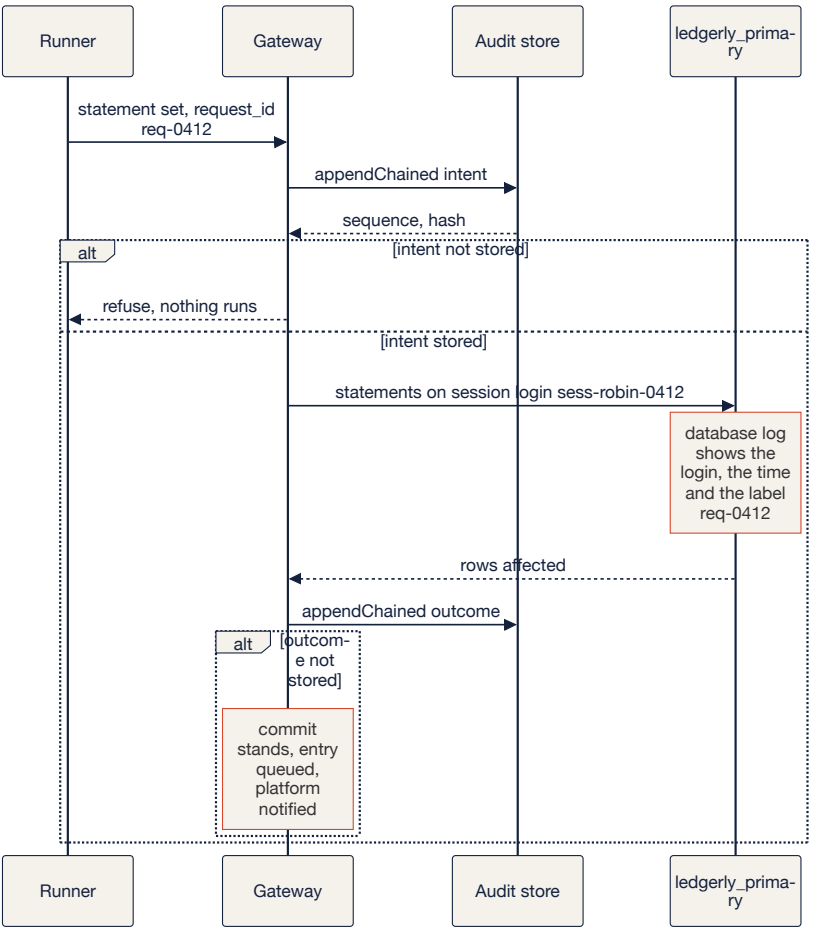
streams silencieux : un agent arrêté laisse un stream sans entrées, ce qui soulève un constat à part entière.

Joindre trois enregistrements

L'enregistrement de la gateway, le log de la base de données et `audit_events` sont produits par trois systèmes qui ne partagent jamais de transaction. Les joindre est ce qui transforme les cas de la matinée d'Alex en une chronologie au lieu de suppositions, et les clés de jointure doivent être choisies avant l'incident, parce qu'on ne peut pas les ajouter après coup.

Trois clés font le travail. Le **login** apparaît dans le log de la base de données et, comme un login de session correspond à exactement une personne ou une instance de workload, il relie une ligne de log à un enregistrement de session. L'**id de demande** est écrit par la gateway dans l'enregistrement d'audit et par le runner dans le propre champ `request_id` du stockage d'audit ; il relie l'instruction à l'approbation qui l'a autorisée. Le runner n'écrit jamais dans `audit_events`, ce que la matrice du Chapitre 9 lui refuse. Les lignes de `audit_events` qui portent le correlation id introduit au Chapitre 3 proviennent du chemin produit, où `admin-console` appelle `payments-service`, et elles se joignent à la même demande par cet id. L'**étiquette de connexion** du Chapitre 5 peut porter l'id de demande jusque dans le log de la base de données pour les connexions médiées. Cette étiquette est définie par celui qui ouvre la connexion. En exécution médiée, la gateway l'ouvre avec l'id de demande de la demande approuvée, elle est donc fiable ; dans un session handout, le client de la personne pourrait y mettre n'importe quoi, si bien que le login reste la clé et l'étiquette reste un indice.

Le diagramme montre une correction de données approuvée et l'endroit où chaque enregistrement est écrit ; le runner n'atteint le primary qu'à travers la gateway. Si l'écriture du résultat échoue, le commit est maintenu et la lacune est détectable.



Ce que l'enregistrement ne doit pas contenir, et qui audite les auditeurs

Un enregistrement d'instruction contient ce que les gens ont tapé. Une personne qui recherche un client par numéro de compte a mis le numéro de compte dans l'instruction, et un log qui stocke l'instruction a copié une valeur Restricted dans un endroit sans protection au niveau des colonnes. L'avertissement du Chapitre 12 s'applique pleinement ; la personne qui construit l'enregistrement doit donc décider de ce qui est conservé.

Je stockerais l'instruction en deux parties. Le **fingerprint** est l'instruction dont chaque littéral est remplacé par un paramètre fictif, si bien que `UPDATE invoices SET status = ? WHERE id IN (?)` enregistre la forme et rien d'autre. Le **hachage des littéraux** est un hachage à clé des valeurs

littérales, qui permet à un enquêteur de tester une valeur soupçonnée contre l'enregistrement sans que l'enregistrement la contienne. La clé compte : des littéraux comme un statut ou un court suffixe de compte ont une faible entropie, et un hachage sans clé pourrait être inversé en essayant des candidats. La clé vit avec le compte d'audit, à l'écart des producteurs, et l'outil de l'enquêteur demande une comparaison et jamais la clé.

```
function buildStatementRecord(session, statement, analysis):
    // analysis comes from the gateway's query analysis, not from the
client
    parsed = parse(statement)
    record = { kind: "intent", session_id: session.id,
               login: session.login, principal: session.principal,
               on_behalf_of: session.on_behalf_of,
               roles: session.roles, request_id: session.request_id,
               fingerprint: redactLiterals(parsed),
               literals_hash: keyedHash(auditKey, literalsOf(parsed)),
               tables: analysis.tables,
               columns: classifyColumns(analysis.columns),
               started_at: now() }
    // a statement that cannot be parsed is recorded as a refusal with
no
    // text, never stored verbatim as a fallback
    if parsed is null:
        record.fingerprint = "unparsed"
        record.literals_hash = null
    return record
```

Les dernières lignes comptent le plus. Un fallback qui stocke l'instruction brute quand le parseur échoue conserverait précisément le cas où un littéral a le plus de chances d'apparaître, parce que les instructions inhabituelles sont celles sur lesquelles un parseur échoue. Je ne stockerais rien et je laisserais le refus parler. Le stockage lui-même est classé au moins Confidential, parce qu'il décrit qui regarde quoi, et il a sa propre liste de lecteurs et sa propre rétention. Ce qu'une entreprise peut enregistrer sur son propre personnel est une question pour vos collègues des services juridique et conformité.

Qui audite `ledgerly_access_admin`

`ledgerly_access_admin` est le rôle qui peut créer des logins et accorder n'importe quoi au sein des rôles qu'il administre, utilisé à travers trois logins : l'étape du pipeline qui applique la définition des accès (Chapitre 3),

le credential broker (Chapitre 4) et la gateway (Chapitre 7). Le Chapitre 3 disait que chaque action va dans le log d'audit ; le mécanisme est que chacun des trois est un producteur doté de son propre stream, si bien que l'enregistrement nomme le login qui a agi. Chaque `CREATE LOGIN`, `GRANT` et `REVOKE` qu'il émet est une entrée `event`, écrite avant le changement, avec la demande ou le run qui l'a provoqué.

Cela donne un contrôle qui boucle la boucle. Le catalogue de la base de données fait foi sur les grants qui existent, et le stockage d'audit fait foi sur ce qui a été accordé. Un grant qui apparaît dans le catalogue sans entrée correspondante a été fait en dehors des trois producteurs, et c'est le grant de 2 h du matin que la vérification de dérive du Chapitre 3 a été construite pour trouver. Les deux contrôles sont une seule chose vue par ses deux bouts : la dérive compare le catalogue avec le code, et celui-ci compare le catalogue avec l'enregistrement. Aucun ne répare quoi que ce soit ; tous deux signalent.

Migrations

Le pipeline de migration écrit dans le stockage d'audit avant d'exécuter une migration, comme l'exige le Chapitre 11, puis de nouveau quand elle se termine ou échoue. La raison de cette entrée externe est que l'outil de migration enregistre aussi ce qu'il a appliqué dans une table à l'intérieur de `ledgerly_primary`, et que `ledgerly_migrator` endosse `ledgerly_owner`, qui peut modifier cette table. Un enregistrement que l'acteur peut réécrire est une commodité, pas une preuve. La règle est qu'une migration compte comme appliquée lorsque l'entrée de début externe et le hash du commit existent dans le stockage d'audit, et qu'une ligne de la table des migrations sans entrée externe correspondante est signalée par la même comparaison entre le catalogue et l'enregistrement, comme un changement qui a contourné le pipeline.

Des alertes auxquelles quelqu'un répondra

Un enregistrement répond aux questions après coup. Quelques-unes de ces questions valent la peine d'être posées en continu, et une alerte est la forme qui le fait. Le test que j'appliquerais à chaque règle est de savoir si quelqu'un est nommément désigné pour y répondre et sait quoi faire.

Ledgerly commence avec trois règles, dont chacune exploite quelque chose que l'enregistrement sait et que les autres sources ignorent.

```
function evaluateAlertRules(window):
    for each r in outcomeRecords(window):
        // 1. large reads outside the person's working hours
        if isReadClass(r.statement_class) and isHuman(r.principal)
           and (r.rows_returned > nightRowThreshold or r.outcome ==
"row-cap-closed")
           and isOutsideHours(r.started_at, r.principal):
            notify("security", "large read outside hours: " +
r.statement_id)
        // 2. break-glass in the role set without a declared request
        if "human_breakglass" in r.roles and not
hasDeclaredRequest(r):
            notify("security", "break-glass role without a
declaration: " + r.session_id)
        // 3. Restricted columns returned to a human principal
        if touchesRestricted(r.columns) and r.rows_returned > 0
           and isHuman(r.principal) and not isDeclaredBreakGlass(r):
            notify("security", "Restricted data read by a human: " +
r.statement_id)
    for each s in sessionsOpened(window):
        if "human_breakglass" in s.roles:
            notify("security", "break-glass in use: " + s.id)
```

Les seuils sont des illustrations ; le bon `nightRowThreshold` est celui qui met une poignée d'événements par semaine devant une personne. La deuxième règle est le contrôle indépendant de la déclaration du Chapitre 8 : ce chapitre alerte quand une personne déclare l'urgence, et cette règle trouve une session qui détient le rôle sans déclaration, c'est-à-dire un grant qui l'a contournée. La troisième règle ne devrait presque jamais se déclencher. `human_readonly` ne peut pas lire les colonnes Restricted et une demande JIT ne peut pas en nommer une ; une lecture humaine de données Restricted est donc soit une session break-glass déclarée, soit un grant qui ne correspond pas à la matrice, et le second cas est un constat de conception.

Chaque règle a besoin d'un responsable nommé et d'une première action : le contact sécurité demande quelle était la tâche, l'ingénieur plateforme d'astreinte confirme la déclaration et ouvre l'élément de revue, l'équipe plateforme révoque un grant qui a contourné le processus. Une règle dont on accuse réception sans y jeter un œil relève du théâtre de sécurité.

La revue des accès

Les alertes surveillent des événements. La **revue des accès** est le contrôle planifié qui vérifie que l'accès de chaque principal est toujours nécessaire, et c'est le contrôle qui attrape la défaillance lente qu'aucun événement ne montre : un accès qui était juste au moment où il a été accordé et qui a discrètement cessé de l'être. Les gens changent d'équipe, un service cesse de lire une table, un rôle garde un privilège que son workload n'a pas utilisé depuis des mois. Le Chapitre 3 disait qu'un grant erroné, correctement déclaré, passe la vérification de dérive, et que c'est dans la revue des accès que ce jugement a sa place.

La revue compare ce qui est accordé avec ce qui a été utilisé. L'enregistrement fournit la seconde liste, et c'est ce qui en fait une décision et non un formulaire à signer.

```
function runAccessReview(period):
    proposals = []
    for each grant in declaredGrants():
        // role to table privileges, identity provider group to role,
        // and the logins of ledgerly_access_admin
        use = lastUse(grant, inRecords: true)
        if use is null or use.at < now() - period:
            proposals.append({ grant: grant, finding: "unused",
                               proposal: "revoke" })
        if grant.owner is null or not isActivePrincipal(grant.owner):
            proposals.append({ grant: grant, finding: "no owner",
                               proposal: "assign or revoke" })
    for each r in breakGlassUses(period):
        if reviewFor(r.id) is null:
            proposals.append({ grant: r, finding: "no review",
                               proposal: "review now" })
    // the review proposes; a named reviewer decides, and is never
    // the person whose access is under review
    return assignReviewers(proposals)
```

Trois règles gouvernent la façon dont j'utiliserais sa sortie. Premièrement, la fonction propose et ne révoque jamais, pour la raison donnée au Chapitre 3 : la révocation automatique d'un grant dont quelqu'un dépend est une nouvelle manière de casser la production. Deuxièmement, la `period` doit être plus longue que le cycle le plus long du workload. Un grant utilisé par un rapport trimestriel est inutilisé la plupart des mois, si bien qu'une revue mensuelle proposerait de le révoquer deux mois sur trois. Troisièmement, l'usage est rapproché par privilège et pas seulement par

table. Un rôle qui fait un select sur une table chaque nuit et ne l'a jamais mise à jour est candidat à une restriction de `S, U` à `S`, et l'enregistrement dispose de ce qu'il faut, parce que chaque enregistrement porte la classe de l'instruction et les tables qu'elle a touchées.

La revue a une seconde entrée que l'enregistrement ne peut pas fournir : ce qui n'est pas un grant. Le registre des copies du Chapitre 13 a sa place à l'ordre du jour, tout comme la liste des personnes qui peuvent approuver les demandes JIT et la liste des personnes qui peuvent lire le stockage d'audit. La question est la même pour chacune : la personne ou le système en a-t-il encore besoin, et quelqu'un en est-il encore responsable ? L'annexe contient un modèle.

Une preuve, pas la conformité

Les régimes de protection des données et ceux des cartes de paiement tendent à poser une famille de questions sur l'accès : qui peut atteindre les données, qui a fait quoi, combien de temps dure l'accès, où se trouvent les copies, et ce qui se passe en cas d'urgence. Un auditeur qui travaille dans le cadre de l'un de ces régimes voudra généralement voir qu'un contrôle existe, qu'il s'est exécuté et que ses résultats ont été examinés. L'interprétation de tout régime relève de vos collègues des services juridique et conformité. Cette section ne revendique rien de plus que ce que l'ingénierie ci-dessus peut livrer, c'est-à-dire une preuve : des enregistrements qui répondent à des questions, et un moyen de montrer qu'ils n'ont pas été altérés.

Rien dans ce chapitre ne rend une entreprise conforme à quoi que ce soit. Le tableau liste les questions qu'un auditeur tend à poser, le contrôle qui répond à chacune, l'artefact qu'il laisse et le chapitre qui le construit ; l'annexe le reprend sous forme de correspondance.

Question	Contrôle	Artefact	Chapitre
Qui peut atteindre les données de production, et par quel chemin ?	La gateway comme seule porte, le schéma de rôles en tant que code, la vérification de dérive	Liste des règles réseau, définitions des rôles, rapport de dérive	3, 7

Question	Contrôle	Artefact	Chapitre
Qui a exécuté une instruction donnée, avec quels grants et quelle approbation ?	`access_audit_record` avec l'id de demande et l'approbateur	L'enregistrement et le rapport de `verifyChain`	14
Combien de temps dure un accès élevé, et qui l'a approuvé ?	Demande JIT avec approbateur et `expires_at`, job de révocation	Enregistrements des demandes, log des révocations, vérification des grants échus	8
Comment les données sensibles sont-elles identifiées et limitées ?	Classification dans le schéma, masquage, colonnes retenues	Le catalogue, la vérification du chargement masqué	12
Où se trouvent les copies des données de production, et qui est responsable de chacune ?	Registre avec responsable et date de fin, sweeper	Registre, log du sweeper	6, 13
Qui modifie le schéma, et qui l'a revu ?	Pipeline de migration, branche protégée, entrée de début externe	Enregistrements de migration rapprochés des commits et des entrées d'audit	11
L'accès est-il encore nécessaire ?	Revue des accès avec relecteurs nommés	Revue terminée avec décisions	14
Que se passe-t-il en cas d'urgence ?	Break-glass avec alerte et revue obligatoire	Enregistrements des revues, historique des alertes	8, 15
Qui administre l'accès lui-même ?	Logins de `ledgerly_access_admin` en tant que producteurs séparés	Leurs streams et la comparaison entre le catalogue et l'enregistrement	3, 14

Compromis et coûts

L'enregistrement a une facture, et l'essentiel se paie à deux endroits.

Le stockage et l'attention. Un enregistrement par instruction, deux fois, croît avec le trafic, et le stockage a besoin de sa propre rétention, de ses propres sauvegardes et de sa propre liste de lecteurs. Le coût le plus important est l'attention. Un stockage que personne n'interroge est un

théâtre de sécurité coûteux ; l'échantillon hebdomadaire des sessions élevées du Chapitre 7, les alertes et la revue font donc partie du prix et ne sont pas des extras.

Une nouvelle dépendance sur le chemin critique. Comme l'intention est écrite avant l'exécution de l'instruction, une panne de l'audit bloque le travail privilégié. L'alternative, exécuter les instructions et journaliser plus tard, rend l'enregistrement facultatif précisément au moment où il compte. Je garderais le refus pour les grants, le break-glass et les écritures, et j'envisagerais une écriture mise en file pour les lectures `human_readonly` de routine, avec un plafond sur la durée pendant laquelle la file peut croître, si l'équipe décide qu'une panne de lecture coûte plus cher qu'une lacune. C'est une décision que l'équipe plateforme doit prendre une fois et consigner par écrit.

Quand ne pas tout construire : une entreprise avec une poignée d'ingénieurs, une seule base de données et pas encore de gateway ne devrait pas commencer par un service d'append, une ancre et un compte séparé. Commencez par trois choses : journaliser la session de chaque personne sous son propre login, envoyer le log de la base de données hors de l'hôte vers un endroit d'où les administrateurs ne peuvent pas supprimer, et charger une personne nommée d'une revue mensuelle. Ajoutez la chaîne et l'ancre quand le nombre de personnes qui pourraient altérer le log fait de la « confiance » une réponse insuffisante.

Erreurs courantes

- **Traiter le log de la base de données comme le log d'audit.** C'est la source la plus complète, et elle se trouve sous le contrôle des administrateurs. Envoyez-le ailleurs et utilisez-le comme une des entrées de la jointure, pas comme l'enregistrement.
- **Une chaîne de hachages sans stockage séparé.** La chaîne ressemble à une preuve et peut être recalculée par quiconque peut écrire dans le stockage. Placez le stockage dans un autre compte, faites détenir la chaîne par le service d'append et ancrez le head.
- **Stocker l'instruction brute.** C'est tentant parce que cela facilite l'enquête, et cela copie des valeurs Restricted dans un stockage sans pro-

tection des colonnes. Stockez le fingerprint et un hachage à clé, et refusez le fallback vers le texte brut.

- **Des alertes sans responsable.** Un canal auquel personne n'est nommément chargé de répondre devient du bruit. Donnez à chaque règle un responsable et une première action.
- **Tout auditer sauf les auditeurs.** Le rôle qui accorde les accès est celui dont les actions comptent le plus. Donnez-lui ses propres producteurs et comparez son travail avec le catalogue.

Points clés

- Le log d'audit est l'enregistrement de qui a fait quoi à quelles données, conservé là où l'acteur ne peut pas le modifier ; l'enregistrement de la gateway, le log de la base de données et `audit_events` en sont trois entrées, pas la chose elle-même.
- Un enregistrement d'instruction se construit à partir d'une liste de questions écrites d'abord, et ses champs d'identité aboutissent à une personne même quand une machine a agi, grâce à `principal` et `on_behalf_of`.
- La détection d'altération vient d'un compte séparé, de producteurs en écriture seule, d'un service d'append qui détient la chaîne et d'une ancre détenue par quelqu'un d'autre, et elle fournit une preuve, jamais une prévention.
- Une chaîne montre que les entrées présentes n'ont pas été altérées ; des contrôles séparés cherchent donc les trous de séquence, les intentions sans résultat, les logins de base de données sans origine connue et les producteurs devenus silencieux ; les trois enregistrements se joignent sur le login et l'id de demande, et l'étiquette de connexion n'est qu'un indice.
- L'enregistrement ne doit jamais devenir une copie des données qu'il décrit ; les instructions sont donc stockées sous forme de fingerprints avec un hachage à clé des littéraux, et le stockage est classé et restreint comme les données.
- Les alertes ont besoin d'un responsable et d'une première action, et la revue des accès compare les grants à l'usage sur une période plus

longue que le cycle du workload, afin que les révocations soient proposées à partir de preuves.

- Une preuve n'est pas la conformité : la correspondance entre question, contrôle, artefact et chapitre montre ce que le système peut prouver, et ce que cette preuve signifie dans le cadre d'un régime quelconque relève de vos collègues des services juridique et conformité.

L'enregistrement existe désormais ; il est joint et surveillé. Reste le jour où l'on en a besoin sous pression, quand la personne qui pose la question n'est plus un auditeur mais un ingénieur d'astreinte face à un credential qui a fuité. Le chapitre suivant passe à l'accès pendant les incidents, et utilise l'enregistrement construit ici pour découvrir ce qu'un principal compromis a réellement touché.

Chapitre 15 — L'accès pendant les incidents

Imaginez Ledgerly un samedi après-midi. Sam, du support, écrit dans le canal d'astreinte : un ordinateur portable professionnel a été volé dans une voiture garée, et une session de la gateway était peut-être ouverte dessus. Robin est d'astreinte et lit le message deux fois. Rien n'indique qu'un problème existe, mais « était peut-être ouverte » est le genre de phrase à laquelle il faut répondre, pas qu'il faut soupeser.

Robin a besoin de trois choses, et l'ordre compte. La première est d'arrêter tout ce que cette session, et tout login derrière elle, peut encore faire. La deuxième est d'apprendre ce qu'elle a fait tant qu'elle le pouvait. La troisième est de maintenir l'activité pendant que les deux premières ont lieu, ce qui signifie que d'autres personnes ont toujours besoin de la base de données, certaines en urgence, et que les approbateurs habituels sont chez eux.

Ce chapitre traite l'accès pendant les incidents comme quelque chose à concevoir et à répéter. Il met en jeu les trois piliers à la fois : l'autorisation quand des grants sont retirés ou ajoutés dans la précipitation, l'exécution quand des réparations doivent s'exécuter sous pression, et l'imputabilité quand l'enregistrement doit dire ce qui s'est passé. Dans le modèle en cinq parties, l'Identité trouve les logins, le Grant et le Chemin les coupent, l'Exécution porte les écritures d'urgence, et l'Enregistrement mesure les dégâts. Ce chapitre ajoute la dernière étape de l'état cible de Ledgerly : l'accès pendant les incidents dispose d'un chemin répété, et la révocation est une routine. La question est de savoir comment Ledgerly arrête l'accès, mesure ce qui a été atteint et continue de fonctionner, sans rouvrir les portes que les chapitres précédents ont fermées.

Le problème, et pourquoi il est difficile

Un incident est le moment où chaque contrôle antérieur ressemble à un obstacle. L'approbation qui était raisonnable un mardi calme devient une attente un samedi, la gateway qui enregistre tout est une chose de plus qui peut tomber en panne, et le raccourci le plus propre à portée de main est un login qui contourne tout cela. La réponse habituelle consiste à construire ce login à l'avance « pour les urgences ». C'est de nouveau le mot de passe admin partagé, sous un meilleur nom.

Trois choses changent sous pression. Le temps est court, si bien que les étapes dont une personne doit se souvenir sont sautées. La connaissance est partielle, si bien que la première estimation du périmètre est fausse dans un sens ou dans l'autre. Et les acteurs sont inhabituels : des personnes qui touchent rarement la base de données doivent soudain le faire, et celles qui la touchent tous les jours sont fatiguées.

La question de conception est donc de savoir quels contrôles se relâchent et lesquels ne se relâchent pas. Je tiendrais une règle : l'urgence change la vitesse, jamais l'identité, jamais l'instruction qui s'exécute, et jamais l'enregistrement. Les délais d'attente et l'ordre des approbateurs peuvent se réduire. Un principal qui agit sous son propre login, une approbation liée au hachage d'une instruction et une entrée dans le log d'audit ne le peuvent pas. Si un contrôle ne peut pas être raccourci sans supprimer l'un de ces trois éléments, c'est le contrôle qui est faux, pas la règle.

La vitesse doit être construite. Une révocation qui demande une douzaine d'étapes manuelles sera faite dans le mauvais ordre ou à moitié. L'ordre est du code, la requête de périmètre est du code, et la réconciliation qui dit si l'incident a laissé des accès derrière lui est du code. Un runbook qui renvoie vers eux est une procédure ; un runbook qui les remplace par du jugement est un espoir.

Confiner une compromission

Le **confinement** désigne ici le fait d'arrêter l'accès d'un principal, que la cause soit un ordinateur portable perdu, un credential qui a fuité ou un workload qui se comporte d'une manière que personne n'attendait. Il ne tranche pas la cause. Le Chapitre 4 a séparé deux questions que la panique

tend à fusionner : comment empêcher un credential de fonctionner, et ce qu'il a fait pendant qu'il fonctionnait. Le confinement répond à la première, et l'ordre ci-dessous permet de continuer à répondre à la seconde.

L'ordre est le suivant : révoquer les grants, fermer les sessions, faire la rotation des credentials, délimiter les dégâts à partir du log d'audit, puis enregistrer et notifier. Chaque étape est à sa place pour une raison.

La révocation des grants vient en premier parce qu'une session sans privilèges ne peut rien faire, et qu'une reconnexion n'obtient rien. La fermeture des sessions vient en deuxième, pour le travail en cours et les connexions ouvertes qui survivent à un grant révoqué (le Chapitre 4 a fait la même remarque à propos des credentials expirés). La rotation vient en troisième parce qu'elle arrête les nouveaux logins et ne fait rien pour ceux qui sont déjà ouverts. La délimitation du périmètre se fait en lecture seule et a besoin de l'enregistrement, pas de l'accès ; elle attend donc que l'accès ait disparu. La notification vient en dernier, mais pas l'enregistrement de chaque étape : chaque étape est ajoutée par un append dès qu'elle se termine.

```
function containCompromise(incident, principal, since):
    unfinished = []
    // every step runs even if the one before failed: a stuck revoke
must
    // not leave sessions open and the credential valid
    for each step in ["revoke-grants", "close-sessions", "rotate-
credentials"]:
        outcome = "done"
        try:
            runContainmentStep(step, principal)
        on StepFailed:
            outcome = "failed"
            unfinished.append(step)
            notify("platform", "containment step failed: " + step + "
" + incident.id)
        // the log never delays containment; an unreachable log queues
the entry
        auditOrQueue({ action: "contain", actor: incident.commander,
                       subject: principal, request: incident.id,
                       step: step, outcome: outcome })
    // read-only and last: it needs the record, not the access
    scope = blastRadius(principal, since)
    saveIncidentRecord(incident.id, { principal: principal, since:
since,
                                      scope: scope, unfinished:
unfinished })
    notify("security", "contained, scope attached: " + incident.id)
    return unfinished
```

Deux détails méritent une remarque. La fonction renvoie les étapes inachevées et ne lève pas d'exception, parce qu'un principal à moitié confiné est un état à suivre, et qu'une exception mettrait fin au seul processus qui en a connaissance. Et `auditOrQueue` est le même helper qu'au Chapitre 7 : une panne du log retarde l'entrée et jamais le retrait de l'accès. L'asymétrie est celle que le Chapitre 8 a choisie pour la révocation, un grant sans entrée de log étant une exposition, et une révocation sans entrée une lacune qui peut être comblée plus tard.

Ce que signifie chaque étape dépend de la nature du principal. Le tableau liste les quatre cas que Ledgerly connaît.

Principal	Révoquer les grants	Fermer les sessions	Faire la rotation des credentials
Personne, logins de session	Retirer les triplets JIT, `human_breakglass` et `human_readonly` de chaque login de session que la personne détient	`endSession` pour chaque login ; désactiver la personne dans le fournisseur d'identité pour qu'aucune nouvelle session ne s'ouvre (Chapitre 7)	Réinitialiser les credentials et les facteurs de la personne dans le fournisseur d'identité ; les logins de session sont supprimés, il n'y a donc aucun mot de passe de base de données à changer
Service, credential dynamique	Mettre fin au lease ; le login disparaît avec lui (Chapitre 4)	Le broker ferme les sessions en mettant fin au lease	Rien à la main ; la demande suivante obtient une nouvelle valeur
Service, credential statique dans une paire A/B	Retirer immédiatement le rôle `app_*` du login qui a fuité, sans attendre que l'ancien login se vide	Fermer ses sessions restantes	Publier une nouvelle valeur sur le login inactif et basculer le pointeur ; redémarrer les services qui ne peuvent pas relire leur secret
Login `ledgerly_access_admin`	Retirer ses privilèges, ce qui arrête le provisionnement pour le principal qui l'utilisait	Fermer ses sessions	Émettre un nouveau credential pour ce principal ; la question du périmètre devient celle des grants et des logins qu'aucune demande n'explique (la vérification de dérive du Chapitre 3)

La ligne statique est celle qui correspond au Chapitre 4. Un credential statique qui a fuité n'est pas une rotation planifiée, et le login qui a fuité est coupé avant que son remplaçant soit prêt, ce qui coûte une panne dans les services qui l'utilisent. Le Chapitre 4 a déjà accepté ce coût, et je maintiendrais cette acceptation : un credential qui a fuité et qui continue de fonctionner est le problème le plus grave. Les services qui ont besoin d'un redémarrage devraient figurer sur une liste avant le jour où cela compte. Dans l'état initial, où chaque service partage un seul login admin, la première étape est une panne de toute l'entreprise, ce qui est l'argument le plus fort pour construire d'abord la révocation de routine.

La dernière ligne est la plus lourde. Le login qui administre les accès peut accorder n'importe quoi au sein des rôles qu'il administre ; le confiner empêche donc la gateway, le pipeline ou le broker de provisionner quoi que ce soit jusqu'à ce qu'un nouveau credential existe. C'est le prix à payer pour couper le login dont la portée est la plus large, et une équipe devrait décider à l'avance qui peut le payer.

Délimiter le rayon d'impact à partir de l'enregistrement

Une fois l'accès disparu, la question devient ce qu'il a fait. La réponse est une seule requête sur `access_audit_record`, dont le Chapitre 14 décrit la conception, exécutée avec les champs que ce chapitre définit. La requête donne le rayon d'impact d'un principal depuis le moment T où son accès a pu être détourné pour la première fois : chaque table qu'il a touchée, combien de lignes sont revenues, combien ont été modifiées, et lesquelles de ces tables contiennent des données Restricted.

```
// illustrative notation; field names as in Chapter 14
SELECT t.table_name,
       classOf(t.table_name)                  AS data_class,
       COUNT(*)                               AS statements,
       SUM(r.rows_returned)                   AS rows_returned,
       SUM(r.rows_affected)                   AS rows_affected,
       SUM(CASE WHEN r.outcome = "refused" THEN 1 ELSE 0 END) AS
refused,
       MIN(r.started_at)                      AS first_seen,
       MAX(r.ended_at)                        AS last_seen
FROM access_audit_record r
     EXPAND r.tables AS t                     // one row per table
touched
WHERE r.kind = "outcome"                   // one row per statement;
intent rows have no counts
  AND r.login IN :logins
  AND r.started_at >= :since
GROUP BY t.table_name
ORDER BY (data_class = "Restricted") DESC, rows_returned DESC;
```

Le choix de `:logins` et de `:since` concentre l'essentiel du jugement. Pour une personne, la gateway crée un nouveau login à chaque session ; l'ensemble est donc constitué de chaque login de session que le principal a eu depuis T, et non d'un seul. Pour T, prenez le moment le plus précoce où l'accès a pu être détourné, comme le dernier moment où l'on sait que l'ordinateur portable était entre les mains de son propriétaire, et non le moment où quelqu'un s'en est aperçu. Une fenêtre trop étroite produit une réponse nette à la mauvaise question.

Le résultat ne vaut que ce que vaut l'enregistrement ; trois contrôles passent donc avant que quiconque le lise.

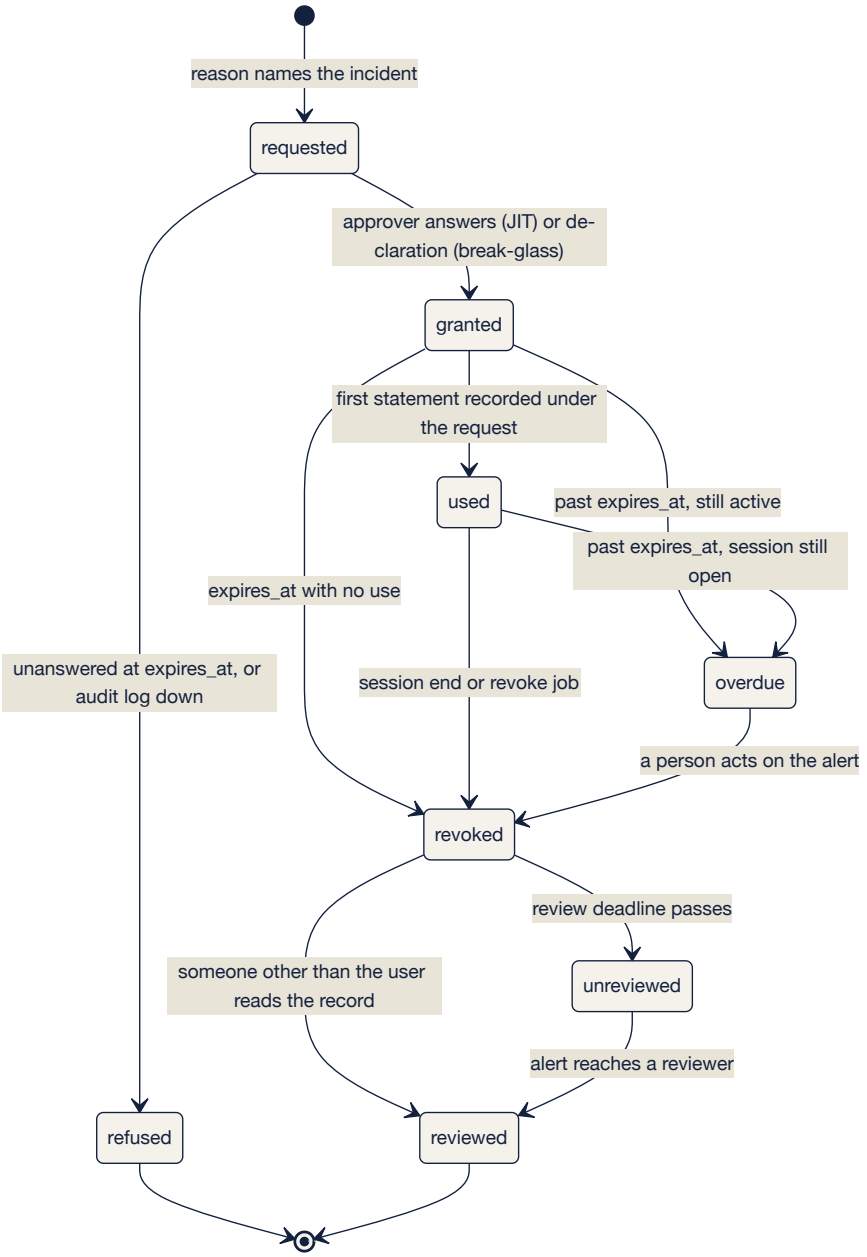
1. `verifyChain` réussit sur la fenêtre, tel que le Chapitre 14 le définit. S'il échoue, le résultat est la preuve d'autre chose que l'incident, et l'incident reçoit un second enregistrement.
2. La fenêtre ne contient aucun trou de séquence et aucune session de la gateway sans session de base de données, ni l'inverse, ce qui correspond à la détection des lacunes du Chapitre 14. Un trou à l'intérieur de la fenêtre est une période où la réponse est inconnue.
3. Les chemins de traverse du principal sont contrôlés séparément : les restaurations, les exports et les copies qu'il a demandés (Chapitre 13), trouvés grâce au registre des copies et non par cette requête.

Si un contrôle échoue, le résultat est une borne inférieure, et je l'écrirais dans l'enregistrement de l'incident en ces termes. La requête a aussi ses propres limites. `rows_returned` est un nombre, pas une liste : il vous dit qu'une table Restricted a renvoyé un certain nombre de lignes dans la fenêtre, et il ne peut pas vous dire de quels clients il s'agissait. Les champs `fingerprint` et `literals_hash` vous permettent de demander si une instruction connue a été répétée, mais les littéraux sont masqués par conception. L'enregistrement borne donc l'exposition et ne l'énumère pas. Un `rows_returned` non nul sur une table Restricted est la ligne dont les contacts sécurité et vos collègues des services juridique et conformité ont besoin, et l'interprétation de ce qu'elle signifie, y compris toute obligation de notification, leur appartient.

La vie de l'accès pendant un incident

Le confinement retire des accès. Un incident a aussi besoin d'en ajouter : quelqu'un doit regarder, et quelqu'un devra peut-être réparer. Le risque dans ce sens est le grant créé dans la précipitation et toujours là le lundi. Le remède est un cycle de vie doté d'une fin visible, dans lequel chaque accès créé pour un incident passe par les mêmes cinq états.

Le diagramme est une vue sur les enregistrements que le Chapitre 8 conserve déjà, pas un nouveau champ de statut. `granted` correspond à `approved` dans l'enregistrement de la demande, `used` signifie qu'une instruction a été enregistrée au titre de la demande, et `reviewed` signifie que quelqu'un d'autre que l'utilisateur a lu l'enregistrement. Repérez les deux sorties qui n'atteignent pas `reviewed` : un grant qui reste au-delà de son terme, et une revue qui n'a jamais lieu.



Pour que cela fonctionne, chaque demande faite pendant un incident porte l'identifiant de l'incident dans sa raison. Le formulaire de la gateway peut l'exiger, et une demande qui n'en porte pas pendant un incident ouvert est refusée. Cette seule convention est ce qui permet à un job de demander « tout ce que cet incident a accordé ».

```
function reconcileIncidentAccess(incident):
    // watches the incident's own requests; never changes access
(Chapter 8:
    // an automatic revoke of unknown grants is a way to break
production)
    for each r in requestsWithReason(incident.id):
        if r.status == "approved" and r.expires_at < now() -
graceMinutes:
            notify("platform", "incident grant past expiry: " + r.id)
        if r.status == "revoked" and reviewFor(r.id) is null
                and now() - r.expires_at > reviewDeadline:
            notify("security", "incident access not reviewed: " +
r.id)
    // the database side: grants that no request explains
    for each g in unexplainedGrants(since: incident.opened_at):
        notify("platform", "grant with no request: " + g.login)

function closeIncident(incident):
    reconcileIncidentAccess(incident)
    // an incident is not over while any access it created is still
open
    if openFindings(incident.id) > 0:
        raise IncidentHasOpenAccess
    markClosed(incident.id)
```

Ce job ne remplace pas la vérification des grants échus ni la vérification de dérive des chapitres 8 et 3. Il regarde les mêmes faits du côté de l'incident, et il donne à l'incident un point de passage : personne ne peut le marquer comme clos tant qu'un grant est encore actif ou qu'un usage break-glass n'a pas été revu. Une équipe qui utilise un système de tickets peut relier `closeIncident` au ticket, ce qui transforme « nous avons oublié de révoquer » en blocage visible.

Revu

Le dernier état est celui que l'on saute le plus souvent, parce que l'incident semble terminé. La revue est celle que le Chapitre 8 décrit pour le break-glass, étendue à chaque grant créé par l'incident. Elle est sans blâme et porte sur le système : l'accès était-il le plus réduit possible pour faire le travail, et existait-il un chemin plus rapide qui aurait dû exister ? L'Annexe contient un modèle de revue post-incident. Le constat qui rapporte est généralement un chemin manquant.

Les écritures pendant un incident

La réparation est l'endroit où il est le plus tentant de relâcher un contrôle, parce que le dommage est visible et que les données sont fausses en ce moment même. Le Chapitre 10 a établi qu'une réparation est une correction de données, et le Chapitre 8 a établi que le break-glass n'est pas une voie pour les réparations. Sous pression, je maintiendrais ces deux décisions et je rendrais rapide le chemin conforme, au lieu de rendre disponible le mauvais chemin.

Le premier choix est de savoir à quel chemin appartient le besoin. L'ordre des contrôles va de la porte la plus étroite à la plus large.

```
function chooseIncidentWritePath(need):
    // first match wins; Restricted values are never written by a
person
    if need.touches_restricted_column or need.is_owner_link_change:
        return "product path: admin-console asks payments-service
(Chapters 9, 10)"
    if need.is_schema_change:
        return "pipeline, expedited lane (Chapter 11)"
    if need.stops_ongoing_harm and need.is_small_limiting_action:
        // pausing, disabling or flagging: never the repair itself
        return "break-glass session, reviewed afterward (Chapter 8)"
    return "data fix: JIT request, runner, expedited review (Chapters
8, 10)"
```

Le break-glass n'est accessible que pour la petite action de limitation : un flag qui empêche un job d'écrire de mauvaises lignes, un statut qui retient un lot. La réparation durable des données est toujours une correction de données, et pour le même raisonnement qu'auparavant : une réparation exécutée dans une session break-glass brute contourne le seul contrôle qui lit l'instruction avant son exécution, et elle s'exécute là où un `UPDATE` non borné est encore possible (le Chapitre 8 le désigne comme un risque accepté). Pendant un incident, la tentation est plus forte, et le coût d'une seule écriture erronée aussi.

La correction de données garde sa forme. La demande nomme une correction revue et porte un `approved_hash` tiré de l'enregistrement du merge, le runner refuse un script dont le hachage diffère, le grant est dérivé de la correction, et le garde-fou de comptage et le snapshot s'appliquent toujours (chapitres 9 et 10). Ce que l'incident change, c'est la revue.

```
function expediteFixReview(fix, incident, reviewer):
    // relaxes: the waiting period, and who may be asked first
    // never relaxes: a reviewer who is not the author, the hash, the
runner
    if incident.status != "open":
        raise NoOpenIncident
    if reviewer == fix.author:
        raise SelfApprovalNotAllowed
    mergeRecord = mergeFix(fix.id, reviewedBy: reviewer, expedited:
true,
                           incident: incident.id)
    // the hash is read from this record by the gateway, not from the
runner
    queueFollowUpReview(fix.id, reviewer: "platform", due:
nextWorkingDay)
    notify("platform", "expedited fix merged: " + fix.id)
    return mergeRecord
```

Cela reflète la lane accélérée que le Chapitre 11 donne aux migrations : même job, même credential, même enregistrement, calendrier différent. La revue de suivi est l'endroit où le raccourci est remboursé, et c'est aussi ce qui transforme un chemin accéléré en événement comptabilisé. Si une équipe constate que la plupart des corrections d'un trimestre l'utilisent, la lane normale est trop lente pour le travail, et la bonne réponse est d'accélérer la lane normale, pas d'arrêter de compter.

> **Avertissement :** Une correction qui « a besoin » du break-glass est le signal de quelque chose d'autre, comme le dit le Chapitre 10 : un plafond auquel il manque une colonne, un pool d'approbateurs trop réduit pour l'heure, ou une colonne Restricted qui relève du chemin produit. Pendant un incident, ce signal est facile à ignorer, et coûteux à ignorer deux fois.

Les cas que les chapitres précédents ont reportés

Trois situations ont été renvoyées à plus tard par les chapitres précédents. Le tableau donne la décision d'emblée ; les sous-sections en donnent le raisonnement.

Situation	Ce qui s'arrête	Ce qui ne change pas	Décidé à l'avance
La gateway est en panne	L'accès humain aux données (pas de seconde porte)	Les services continuent de fonctionner avec leurs propres credentials	Qui restaure la gateway, et quand l'instrument scellé peut être utilisé
Une restauration est nécessaire pendant l'incident	Rien, si le chemin a été répété	Approbateur, cible isolée, entrée au registre, suppression planifiée	Un approbateur nommé, une cible préparée, un runbook de restauration
Les approbations sont lentes la nuit	Une réparation qui n'est pas une urgence	L'auto-approbation est toujours refusée	Un ordre d'escalade et la personne qui peut élargir le pool

La gateway est en panne

Les chapitres 7 et 8 ont décidé qu'il n'y a pas de seconde porte : les humains n'atteignent la base de données qu'à travers la gateway, si bien qu'une panne de la gateway est aussi une panne du break-glass. Le raisonnement était qu'un credential qui fonctionne sans la gateway fonctionne aussi sans la journalisation, l'identité et l'expiration. Je m'y tiendrais la pire des nuits, et le premier geste lors d'une panne est de remarquer ce qui ne s'est pas arrêté. Les services se connectent avec leurs propres credentials et continuent de fonctionner ; beaucoup d'incidents pendant lesquels la gateway se trouve en panne n'exigent donc aucune personne dans la base de données. Annuler le déploiement, mettre le job en pause, basculer le feature flag, faire passer le service à l'échelle : rien de tout cela ne passe par la gateway.

Si l'incident exige un accès aux données, l'incident devient deux incidents, et le premier est la gateway. La restaurer passe avant la question des données, et ceux qui la restaurent sont l'équipe plateforme, au moyen du chemin de déploiement et de la redondance que le Chapitre 7 a demandés.

Si toutes les instances sont en panne, aucun chemin humain vers les données n'existe, et le seul instrument restant est le superutilisateur que le Chapitre 3 a traité comme scellé. C'est un outil au niveau du moteur, pas un chemin vers les données : il contourne les protections append-only et Restricted, et sa route réseau est fermée aux personnes. Je limiterais son

usage à la restauration de la gateway ou de ce dont la gateway dépend, jamais à la lecture ni à l'écriture de données métier, et je consignerais la règle par écrit.

> **Avertissement :** N'utilisez pas l'instrument scellé pour regarder des données parce que la gateway est en panne. Il contourne les protections append-only et Restricted, et l'accès qu'il donne est précisément celui que chaque chapitre précédent a été construit pour empêcher. Si des données doivent être lues, restaurez d'abord la gateway.

Son usage dans le cadre de la règle a un mécanisme. La récupération du credential laisse un enregistrement, une seconde personne est présente et nommée, l'ouverture de la route réseau est un changement doté d'un responsable et d'une heure de fin, et le credential fait ensuite l'objet d'une rotation parce qu'une personne l'a vu. La gateway n'enregistre rien pendant cette fenêtre ; le log du moteur est donc la seule trace. Une fois la gateway revenue, l'équipe plateforme ajoute une entrée à la main, comme l'entrée d'audit externe que le Chapitre 14 décrit pour les migrations, et l'usage reçoit la revue que reçoit tout usage du break-glass.

Une restauration pendant un incident

Le Chapitre 13 a fait de la restauration une demande : un approbateur qui n'est pas le demandeur, une cible isolée, une entrée au registre, une suppression planifiée et un démarrage fermé. Le coût nommé alors était la vitesse pendant un incident, et sa réponse était de répéter et d'organiser l'approbation à l'avance. C'est là tout le mécanisme ; il vaut donc la peine de préciser ce que signifie « à l'avance ».

Avant l'incident, le runbook nomme qui peut approuver une restauration la nuit, la cible est préparée et isolée, et la demande de restauration dispose d'un formulaire qui accepte un identifiant d'incident. Pendant l'incident, la demande passe sans changement : l'approbateur n'est pas le demandeur, l'entrée au registre vient en premier, la suppression planifiée est définie avant que la copie existe, et les logins sont désactivés et la définition des accès appliquée avant que la copie s'ouvre. L'incident ajoute la raison, pas un raccourci.

Ce à quoi sert la copie compte davantage que la manière dont elle est faite. Pendant un incident, une copie restaurée est une source de lecture, généralement pour d'anciennes valeurs : l'état d'une colonne avant une écriture erronée, ou des lignes supprimées par erreur. Ses lecteurs sont les personnes nommées dans la demande. La réparation qui suit s'exécute sur le primary en tant que correction de données, les valeurs lues dans la copie servant d'entrée à un script revu, et le `reapplyDeletions` du Chapitre 13 s'exécute avant que la copie s'ouvre, afin que la copie ne fasse pas revenir des données supprimées volontairement. Remplacer le primary par une copie restaurée est une décision de reprise, et elle appartient à l'équipe plateforme et au commandant d'incident. La question d'accès qu'elle contient est celle ci-dessus : qui lit la copie, et quand elle est supprimée.

Les approbations lentes la nuit

Le scénario de nuit du Chapitre 8 s'est conclu par une décision que ce chapitre conserve : si le pool d'approbateurs ne répond pas et que le besoin peut attendre, laissez une note et arrêtez-vous. C'est un bon résultat, et un résultat que l'ancienne Ledgerly ne pouvait pas produire. Pour un besoin qui ne peut pas attendre, l'escalade est intégrée à la demande ; un déclenchement d'astreinte resté sans réponse descend donc le pool au sein de la même demande.

```
function escalateApproval(request, pool):
    // one request, one record: the page moves, the request does not
restart
    for each approver in pool.inOrder():
        if approver == request.requester or approver ==
request.fix_author:
            continue
        page(approver, request.id)
        if waitForAnswer(request.id, within: escalationMinutes):
            return
    // pool exhausted, window still open: the incident commander
decides
    notify("platform", "approval unanswered, pool exhausted: " +
request.id)
    // never widens the pool or approves by default; the request stays
pending
    // until expires_at, when it is refused
```

Deux règles encadrent ce mécanisme. Premièrement, le pool n'est élargi que par une personne qui en est responsable, et l'élargissement est un enregistrement : pendant un incident déclaré, le commandant d'incident peut ajouter un approbateur nommé pour l'incident, avec une fin. Deuxièmement, une approbation automatique pour la personne d'astreinte n'est toujours pas construite, parce qu'un contrôle qui accorde chaque demande en quelques secondes n'est pas un contrôle. Si les approbations dépassent régulièrement la fenêtre du grant, le constat porte sur la rotation, la taille du pool ou l'heure, et le remède est d'ajouter des personnes au pool, pas de retirer l'approbateur de la boucle.

Répéter les chemins

Un chemin qui ne s'est jamais exécuté n'est pas testé. Tout ce chapitre est construit à partir d'étapes que personne ne veut découvrir pour la première fois pendant un incident ; chacune a donc besoin d'une répétition, appelée ici game day : un exercice planifié avec un scénario, un résultat attendu et une vérification de l'enregistrement. Quatre scénarios couvrent ce que ce chapitre a construit. Ils sont défensifs par conception, chacun part d'un déclencheur que la surveillance sait déjà voir, et chacun porte une condition de réussite observable.

Scénario	Déclencheur	Comportement attendu	Condition de réussite	Parties mises en jeu
Fuite de credential	Le credential d'un principal d'exercice est déclaré exposé	`containCompromise` dans l'ordre ; requête de périmètre jointe à l'incident	Aucun login actif pour le principal, unfinished est vide, le périmètre nomme chaque table que l'exercice a touchée	Identité, Grant, Enregistrement

Scénario	Déclencheur	Comportement attendu	Condition de réussite	Parties mises en jeu
Gateway en panne	Une puis toutes les instances de la gateway arrêtées dans une copie hors production	L'accès humain s'arrête ; les services continuent de fonctionner ; le chemin de restauration s'exécute	L'équipe plateforme restaure la gateway à l'aide du runbook ; instrument scellé inutilisé, ou utilisé dans le cadre de sa règle avec un enregistrement	Chemin, Enregistrement
Export suspect	Une alerte sur un `rows_returned` élevé, issue de `evaluateAlertRules` (Chapitre 14)	Confiner le principal ; trouver la destination dans le registre des copies ; la supprimer	La copie est trouvée, close et enregistrée ; la requête de périmètre concorde avec le registre	Identité, Enregistrement, Exécution
`UPDATE` massif erroné	Une écriture qui respecte le row cap et modifie les mauvaises lignes, dans un environnement de répétition	Arrêter les écritures suivantes ; calculer l'inversion à partir du snapshot ; l'exécuter comme une correction de données accélérée	Inversion mergée par un relecteur qui n'est pas l'auteur ; revue de suivi mise en file	Exécution, Grant, Enregistrement

Un game day ne réussit que sur le résultat observé, jamais parce que quelqu'un se souvient de ce qu'il faut faire. Si un exercice aboutit parce que quelqu'un connaissait un raccourci, il a échoué, et le raccourci devient la chose à construire. Notez le temps que chaque étape a réellement pris ; ces chiffres appartiennent à votre entreprise, et je ne donnerais pas un chiffre qui ne lui appartient pas.

Utilisez une copie hors production quand c'est possible, et des principals synthétiques dans les exercices en production, afin qu'aucune donnée client réelle ne soit lue pour s'entraîner à la lire. Les copies d'exercice sont enregistrées et closes comme toute autre copie (Chapitre 13).

Compromis et coûts

Le premier coût est la répétition elle-même. Les game days éloignent les ingénieurs du travail sur le produit, et un exercice a besoin d'un responsable et d'un calendrier. Pour une équipe de trois personnes, un seul exercice par an couvrant la fuite de credential est un début raisonnable, et les trois autres scénarios peuvent attendre que l'équipe grandisse.

Le deuxième coût est la panne que provoque le confinement. Révoquer immédiatement le credential d'un service est une panne de ce service, et confiner `ledgerly_access_admin` arrête le provisionnement. Ce sont deux choix à faire à l'avance, avec les personnes qui en supportent le coût, parce que la personne qui décide à 3 h du matin ne devrait pas en même temps découvrir ce que coûte le choix.

Le troisième coût est la convention de l'identifiant d'incident. Elle ajoute un champ à chaque demande et une raison de plus de refuser un formulaire, et elle ne vaut que ce que vaut la volonté des gens de le remplir. La réconciliation qui en dépend échoue silencieusement quand il n'est pas rempli ; la gateway devrait donc imposer le champ plutôt que de compter sur l'habitude.

Le quatrième coût est l'honnêteté sur le périmètre : certains incidents se terminent par une borne plus large que la vérité, et je rapporterais la borne plutôt que de revendiquer une précision que l'enregistrement n'a pas.

N'appliquez pas toute la machinerie à un système qui n'a pas encore de log d'audit. `blastRadius` a besoin de l'enregistrement, et `reconcileIncidentAccess` a besoin des enregistrements des demandes ; une équipe qui ne les a pas devrait d'abord les construire (chapitres 8 et 14), et entre-temps écrire un court runbook qui nomme qui révoque, qui ferme les sessions et qui est responsable de la question de savoir ce qui a été touché.

Erreurs courantes

- **Faire une rotation sans révoquer.** Une rotation change le secret et laisse en place la session ouverte et les privilèges accordés. C'est ten-

tant parce que la rotation est une routine que l'équipe possède déjà. Révoquez d'abord, fermez ensuite les sessions, puis faites la rotation.

- **Délimiter avant de confiner.** L'envie de comprendre passe avant l'envie d'arrêter, et une équipe peut passer une heure sur la requête pendant que l'accès reste actif. Confinez d'abord. L'enregistrement survit au confinement parce qu'il vit dans un stockage que l'accès ne peut pas toucher.
- **Faire confiance au log sans le vérifier.** Une requête de périmètre sur une fenêtre qui contient une lacune se lit comme une réponse nette. Vérifiez d'abord la chaîne et la séquence, et rapportez une borne inférieure quand l'une ou l'autre échoue.
- **Le break-glass comme lane de réparation.** Il est rapide et disponible, et il n'a aucune revue des instructions. Utilisez-le pour la petite action de limitation et exécutez la réparation comme une correction de données accélérée.
- **Laisser ouverts les accès de l'incident.** L'incident semble terminé quand la cause est corrigée, et les grants restent. Liez la clôture de l'incident à la réconciliation, afin que le ticket ne puisse pas être clos tant qu'un accès est ouvert.

Points clés

- L'urgence change la vitesse d'un contrôle, jamais l'identité qui agit, l'instruction qui s'exécute ni l'enregistrement qui reste.
- Le confinement a un ordre : révoquer les grants, fermer les sessions, faire la rotation des credentials, délimiter le périmètre à partir du log d'audit, puis enregistrer et notifier, et chaque étape s'exécute même si la précédente a échoué.
- La révocation n'est pas la rotation : une rotation arrête les nouveaux logins et ne fait rien pour ceux qui sont déjà ouverts, c'est pourquoi un credential statique qui a fuité est coupé d'abord et remplacé ensuite.
- Le rayon d'impact provient d'une seule requête sur `access_audit_record`, et c'est une borne inférieure dès que la chaîne ou la séquence présente une lacune.
- L'accès pendant un incident passe par les états demandé, accordé, utilisé, révoqué et revu, et un incident ne peut pas être clos tant qu'un

grant qu'il a créé est encore ouvert.

- Les réparations pendant un incident sont des corrections de données avec une revue accélérée, et le break-glass reste réservé à la petite action qui arrête le dommage.
- Un chemin qui n'a pas été répété est une hypothèse ; la fuite de credential, la panne de la gateway, l'export suspect et l'`UPDATE` massif erroné ont donc chacun un game day avec une condition de réussite observable.

Le Chapitre 16 rassemble les contrôles de tout le livre en une checklist et une échelle de maturité, et revient à la phrase que chaque chapitre a servie.

Chapitre 16 — Une checklist d'accès et réflexions finales

Reprenons Ledgerly un an après le questionnaire de sécurité du Chapitre 1. Un autre client potentiel envoie les deux mêmes lignes : « Listez toutes les personnes et tous les systèmes qui peuvent se connecter à votre base de données de production » et « Comment savez-vous ce qu'ils ont fait ? » Alex ouvre l'inventaire des accès au lieu du gestionnaire de mots de passe, lit la liste des portes et la renvoie dans l'après-midi. Rien dans la réponse n'est impressionnant. Elle est courte, chaque ligne a un responsable et une fin, et rien n'a demandé de fouiller.

Robin, d'astreinte cette semaine-là, demande sur le chat : « Nous avons une gateway, un runner, un pipeline, un stockage d'audit et trois jobs de revue. Lesquels nous fallait-il, et lesquels avons-nous construits parce que le livre le disait ? » Alex n'a pas de réponse bien rangée. Certaines pièces ont gagné leur place la première fois qu'un grant a expiré de lui-même ; d'autres n'ont jamais servi pour de vrai, et Alex ne sait pas dire si cela signifie qu'elles fonctionnent ou qu'elles restent inutilisées.

Ce chapitre sert les trois piliers à la fois : l'autorisation, l'exécution et l'imputabilité. Il n'ajoute aucun nouveau contrôle. Il ajoute les étapes de l'état cible dans l'ordre où une équipe peut les atteindre, les patterns notés côte à côte, et une checklist dont les lignes sont des observations plutôt que des opinions. La question à laquelle il répond est celle de Robin : comment une équipe décide-t-elle de ce qu'elle construit ensuite, et comment sait-elle qu'un contrôle fonctionne ?

Le problème, et pourquoi il est difficile

Quinze chapitres laissent au lecteur beaucoup de contrôles et aucun ordre. Chacun a été présenté avec son propre raisonnement, et chaque raisonnement tenait dans son chapitre. Lus ensemble, ils ressemblent à un projet d'un an ou plus, et une équipe face à cette liste fait en général l'une de

deux choses. Elle commence par ce qui coûte le moins et s'arrête quand le travail devient politique, ou elle commence par ce qui fait le plus peur et construit un élément sophistiqué de la conception pendant que le mot de passe partagé se trouve encore dans cinq fichiers de configuration.

La difficulté est que « terminé » se voit mal dans le travail sur les accès. Un schéma de rôles qui existe sur le papier et un schéma de rôles appliqué se ressemblent en réunion de revue ; les équipes mesurent donc ce qui a été installé, et « installé » est la mauvaise unité. L'unité utile est un test que quelqu'un peut exécuter : une instruction que l'on peut retracer, une connexion qui est refusée, un grant qui a disparu.

C'est aussi une affaire de construction du système plutôt que de contenu de la politique. Une politique dit ce qui devrait se passer ; une conception détermine ce qui peut se passer et ce qui laisse une trace quand cela se passe. Chaque ligne ci-dessous peut recevoir une réponse en regardant un système, un log ou une table, et une ligne à laquelle on ne peut répondre qu'en demandant à quelqu'un ce qu'il a l'intention de faire ne figure pas dans la liste.

La thèse, concrétisée

Le livre a défendu une seule phrase, et c'est ici qu'on la confronte aux chapitres :

> L'accès aux bases de données en production n'est pas un problème de distribution de credentials ; c'est un problème d'autorisation, d'exécution et d'imputabilité.

La phrase a trois piliers, et chaque pilier repose sur des chapitres précis. L'autorisation, c'est le Grant plus le Chemin, décidés à chaque demande. L'exécution, c'est ce qui arrive à chaque instruction. L'imputabilité, c'est l'Identité plus l'Enregistrement. Un mot de passe ne touche que le premier, et seulement à la porte.

Pilier	Parties du modèle	Chapitres qui le construisent	La question à laquelle le lecteur peut désormais répondre
Autorisation	Grant, Chemin	3, 5, 6, 8, 12	Que peut faire ce principal, où, pour combien de temps, et avec quelles colonnes masquées ?
Exécution	Exécution	7, 9, 10, 11, 13	Comment cette instruction a-t-elle atteint les données, qui ou quoi l'a inspectée, et qu'est-ce qui l'a bornée ?
Imputabilité	Identité, Enregistrement	4, 7, 14, 15	Quelle personne ou quelle instance de workload a agi, à la demande de qui, et quelle preuve la relie au résultat ?

Les chapitres 1 et 2 encadrent les trois piliers, et le Chapitre 15 les met à l'épreuve sous pression. Le Chapitre 7 apparaît deux fois parce que l'access gateway est à la fois le chemin et l'endroit où une personne devient un login de session qu'un log peut nommer ; le Chapitre 13 se range sous l'exécution parce que l'étape d'export de `reporting-job` exécute un ensemble d'instructions, et que les autres portes latérales sont des routes qui contournent les mêmes contrôles.

Pourquoi le travail sur les credentials était nécessaire sans être la réponse

Le Chapitre 4 traite des credentials, et c'est le chapitre que la plupart des lecteurs s'attendent à trouver au centre du livre. Il est nécessaire. Un service qui lit son mot de passe dans un fichier partagé n'a pas d'identité propre, et sans identités distinctes l'enregistrement ne peut nommer personne. Les credentials de courte durée et dynamiques font de la révocation une étape de routine. La workload identity supprime le fichier.

Rien de tout cela ne décide de ce qu'un principal peut faire une fois entré, de la façon dont une instruction est vérifiée, ni de ce qui est consigné. Une équipe pourrait terminer parfaitement le Chapitre 4 et avoir encore un rôle propriétaire de toutes les tables, une session sur ordinateur portable qui valide chaque instruction telle qu'elle est tapée, et un log dont chaque ligne porte un nom distinct sans qu'aucune ne porte de raison. Un stockage de mots de passe fournit les noms distincts dont la partie Identité a besoin et, par la révocation, un morceau du Grant ; un mot de passe ne touche que le premier pilier, et seulement à la porte. Les autres parties se construisent ailleurs, et c'est le reste du livre qui les construit.

Quand une proposition dit « nous allons régler les accès en améliorant X », placez X sur le modèle avant de l'approuver :

```
function placeControl(proposal):
    parts = partsOfModelTouched(proposal)   // Identity, Grant, Path,
Execution, Record
    if "Execution" not in parts and "Record" not in parts:
        // a control that touches only the door leaves two of three
pillars as they were
        note("improves how access is obtained; says nothing about what
it does or what remains")
    for each part in ["Identity", "Grant", "Path", "Execution",
"Record"]:
        if part not in parts:
            note("unchanged: " + part)
    return notes
```

La fonction n'est pas un gate. Elle nomme les parties qu'une proposition laisse en l'état, parce qu'une partie vide est un constat et non un oubli.

Un catalogue de patterns, notés par pilier

Le Chapitre 1 a tracé l'éventail des façons d'entrer, et le Chapitre 7 a comparé les chemins concrets. Le tableau ci-dessous met côte à côte les cinq patterns de cet éventail et note chacun au regard des trois piliers. Les notes relèvent de mon jugement, pas d'une mesure. Elles disent quelle part d'un pilier le pattern peut prouver à lui seul, à grants et journalisation des chapitres précédents égaux. Lisez « fort » comme « le pattern est construit pour prouver cela », « partiel » comme « il en prouve une partie, ou seulement si l'on ajoute autre chose », et « aucun » comme « le pattern ne dit rien ici ».

Pattern	Autorisation	Exécution	Imputabilité
Accès direct (du client à la base de données)	Partiel : grants seulement, la portée est ce que le réseau permet	Aucun	Partiel au mieux : des logins par personne si quelqu'un les provisionne, sinon aucun
Bastion	Partiel : rétrécit le chemin réseau	Aucun : il relaie, il ne distingue pas un `SELECT` d'un `DROP`	Partiel : qui s'est connecté à l'hôte, pas ce qui s'est exécuté

Pattern	Autorisation	Exécution	Imputabilité
Session proxy ou access gateway	Fort : logins par session, expiration, grants par demande	Partiel : borné par les grants et les guarded settings, les instructions sont tapées librement	Fort sur le qui et le quand, partiel sur le quoi
Exécuteur médiateur d'instructions	Fort	Fort : inspecté avant de s'exécuter, lié à un approved hash	Fort : instruction, approbateur et résultat dans un seul enregistrement
Chemin produit (sans SQL)	Fort : l'opération est la permission	Fort : l'opération est fixe	Partiel : la personne figure dans la piste d'audit de l'outil, le log de la base de données montre le login du service

Accès direct

- **Prouve :** qu'un login existe et que ses grants étaient ce qu'ils étaient.
- **Ne peut pas prouver :** quelle personne a utilisé un login partagé, ni ce qu'a fait une instruction.
- **Coût :** nul à construire, et c'est là son attrait. Tous les coûts arrivent plus tard, sous la forme d'une semaine de fouilles.
- **Mode de défaillance :** le mot de passe admin partagé du Chapitre 1.
- **Ne pas utiliser quand :** plus d'une personne ou d'un workload peut atteindre le login. C'est tolérable pour un système à un seul ingénieur dont le seul consommateur est son responsable, et je consignerais ce qui ferait cesser cette situation.

Bastion

- **Prouve :** qu'une connexion est passée par un hôte contrôlé unique et, si les comptes y sont nominatifs, qui s'y est connecté.
- **Ne peut pas prouver :** ce qu'était une instruction, puisque l'hôte ne parle pas le protocole de la base de données.
- **Coût :** un hôte à patcher et à maintenir disponible, en échange d'un endroit unique où couper le chemin réseau.
- **Mode de défaillance :** un compte de bastion partagé refait de l'hôte le mot de passe partagé, un saut plus tôt.

- **Ne pas utiliser quand :** vous avez besoin de preuves au niveau des instructions. Il reste utile comme portée réseau devant quelque chose qui, lui, enregistre.

Session proxy ou access gateway

- **Prouve :** qu'une personne nommée a détenu un login de session précis, entre deux horodatages, sous des grants précis.
- **Ne peut pas prouver :** qu'une instruction tapée dans cette session était une bonne idée. La session est un handout (Chapitre 9), et tout ce que les grants permettent peut être tapé.
- **Coût :** un nouveau système qui doit être disponible quand quelqu'un en a besoin, et le chemin d'approbation qui l'accompagne (chapitres 7 et 8).
- **Mode de défaillance :** une gateway avec une seconde porte à côté, c'est-à-dire un mot de passe partagé sous un meilleur nom.
- **Ne pas utiliser quand :** les tâches se répètent et ont une forme fixe. Elles ont leur place sur un chemin produit, où personne n'a besoin de session.

Exécuteur médiateur d'instructions

- **Prouve :** que cet ensemble exact d'instructions, identifié par son hash, a été approuvé par une personne nommée, a été inspecté, s'est exécuté dans une borne et a produit un résultat compté.
- **Ne peut pas prouver :** que l'analyse a compris l'instruction. L'analyse des requêtes réduit les erreurs et n'est pas la frontière d'autorisation, qui reste aux grants (Chapitre 9).
- **Coût :** chaque tâche attend la revue, rien n'est interactif, et quelqu'un doit construire et maintenir le runner.
- **Mode de défaillance :** une approbation toujours accordée en quelques secondes, c'est-à-dire du théâtre de sécurité avec des étapes en plus.
- **Ne pas utiliser quand :** le travail est exploratoire. Une personne qui cherche à comprendre un état a d'abord besoin d'une session de lecture, et c'est à cela que sert `human_readonly`.

Chemin produit

- **Prouve :** qu'une opération définie s'est exécutée, pour un rôle dans un outil, avec la piste d'audit propre à l'outil.
- **Ne peut pas prouver :** quoi que ce soit sur les tâches pour lesquelles personne n'a construit de fonctionnalité. Les questions libres n'ont pas de chemin.
- **Coût :** une fonctionnalité par tâche, et le cycle de changement le plus lent des cinq.
- **Mode de défaillance :** le chemin produit qui accumule une échappatoire « exécuter n'importe quel SQL » parce que le premier cas urgent n'avait pas de fonctionnalité.
- **Ne pas utiliser quand :** la tâche ne s'est pas encore répétée. Le Chapitre 10 décrit le signal qui permet de transformer une correction répétée en fonctionnalité, et une tâche survenue une seule fois n'est pas ce signal.

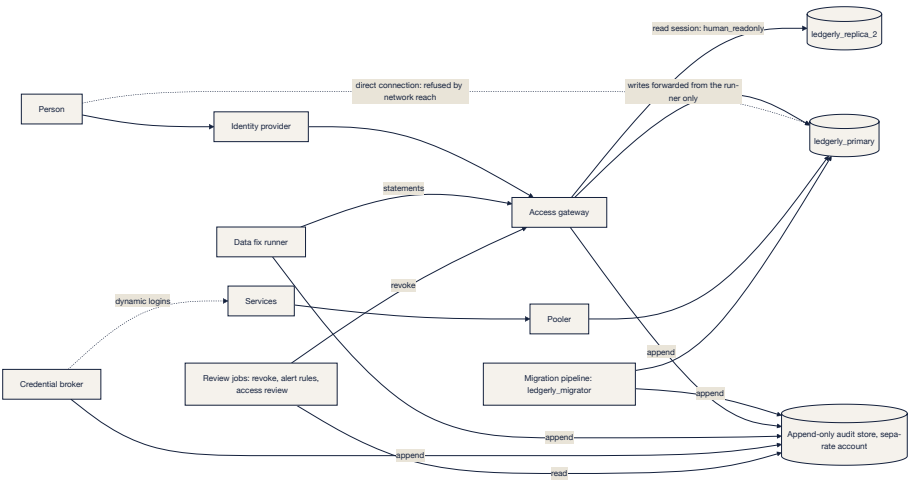
Le pattern le plus à droite n'est pas toujours le meilleur, et un exécuteur médiateur d'instructions dépasse ce dont a besoin une question sur les factures d'un client. Je choisirais tâche par tâche, avec le pattern le moins coûteux qui prouve encore les piliers que la tâche met en jeu :

```
function choosePattern(task):
    // a person who never opens a session cannot leave one open
    if task.repeats and task.hasFixedShape:
        return "product path"
    // a schema change takes the migration pipeline, never the runner
(Chapter 11)
    if task.changesSchema:
        return "migration pipeline"
    // Restricted columns and owner links are changed by the product,
not by SQL
    if task.writesRestrictedColumn or task.changesOwnerLink:
        return "product path"
    if task.writesData:
        // reviewed statement set, approved hash, runner executes
        return "statement-mediating executor"
    if task.readsData:
        return "access gateway: human_readonly on ledgerly_replica_2"
    // anything else is an exception with a named owner and an end
    return "exception: owner, reason, ends_on"
```

La dernière ligne est l'essentiel. Une route qui n'entre dans aucune case va dans l'inventaire avec un responsable et une date de fin, comme le demande le bloc d'inventaire du Chapitre 1, et une exception sans l'un ou l'autre est un reliquat.

L'architecture de référence

Le diagramme montre l'état cible de Ledgerly avec tous les contrôles du livre sur une seule image. Regardez trois choses : aucune flèche humaine n'atteint une base de données sans passer par la gateway, toute écriture venant d'une personne passe par le runner, et le stockage d'audit se trouve dans un compte où les administrateurs de bases de données ne peuvent pas écrire.



L'image simplifie : les répliques de lecture applicatives du Chapitre 6 partagent une boîte avec leur primary, et le chargement masqué vers l'analytics, l'export pour le fournisseur et les sauvegardes (Chapitre 13) sont laissés de côté, parce que chacun est une copie avec un responsable et une expiration, et qu'un dessin portant toutes les copies n'aurait plus de centre lisible.

Chaque boîte correspond au modèle : le fournisseur d'identité et le broker fournissent l'Identité ; les rôles, les logins de session et le job d'expiration fournissent le Grant ; la règle réseau derrière la flèche en pointillés et la gateway fournissent le Chemin ; le runner et le pipeline fournissent l'Exé-

cution ; le stockage d'audit et les jobs de revue fournissent l'Enregistrement. Une boîte qui ne fournit aucune partie est candidate au retrait, et la question de Robin mérite ce test.

Une échelle de maturité avec des tests de sortie

L'état cible est une direction et non une ligne d'arrivée ; les étapes du livre sont donc regroupées ici en cinq paliers. Le regroupement est le mien. Il suit ce qu'une équipe peut changer avant que le changement suivant devienne plus facile, et d'autres organisations le réordonneront ; le seul ordre que je défendrais est que l'inventaire vient en premier, parce que sans lui personne ne sait ce que chaque étape ultérieure doit couvrir. La tenue des enregistrements gagne aussi à commencer tôt sous une forme simple, puisque chaque palier ultérieur se juge plus facilement avec des preuves.

Un palier est terminé quand son test de sortie passe, et « nous l'avons installé » n'est pas un test. Chaque test est une observation que quelqu'un peut répéter.

Palier	Ce qu'il ajoute	Test de sortie	Pourquoi les équipes calent ici
1. Connaître les portes	Inventaire des principals, des portes, des copies (chapitres 1, 2, 13)	Un document liste chaque porte avec un responsable et une fin, et un nouvel ingénieur peut répondre à « qui peut atteindre le primary ? » à partir de lui seul	L'inventaire est traité comme un projet, il n'est donc jamais terminé et jamais digne de confiance
2. Nommer chaque principal	Un login et un rôle propres par principal, des credentials dynamiques, des budgets par rôle (chapitres 3, 4, 5)	Une journée de log de la base de données ne contient aucun login partagé, et aucun service ne se connecte comme le propriétaire ou l'admin	La bascule ressemble à un risque de panne pour chaque service, si bien que le mot de passe partagé reste « pour l'instant »
3. Un seul chemin, borné	Gateway, répliques, JIT et break-glass, classification et masquage (chapitres 6, 7, 8, 12)	Une connexion directe au primary depuis un ordinateur portable échoue, un grant JIT disparaît à `expires_at` sans que personne n'agisse, et une session de lecture ne renvoie aucune colonne Restricted	La peur de la gateway comme point unique de défaillance, et des approbateurs qui ne peuvent pas répondre la nuit

Palier	Ce qu'il ajoute	Test de sortie	Pourquoi les équipes calent ici
4. Médié et consigné	Politique d'exécution, runner, corrections relues, migrations par le seul pipeline (chapitres 9, 10, 11)	Une écriture depuis le client propre d'une personne est refusée, et le runner refuse un ensemble d'instructions dont le hash diffère de `approved_hash`	La revue devient le goulot d'étranglement, si bien que les exceptions s'accumulent plus vite que les fonctionnalités produit
5. Prouvable et répété	Stockage d'audit chaîné par hachages, règles d'alerte, revue des accès, game days (chapitres 14, 15)	Une instruction choisie sans prévenir parmi celles d'hier peut être retracée jusqu'à une personne, une demande et un résultat sans interroger personne ; `verifyChain` passe ; un game day se termine avec la révocation des accès confirmée par l'enregistrement	Le log existe et personne ne le lit, et le game day est la première chose supprimée quand une release est en retard

Un palier peut échouer à son test après avoir été déclaré terminé, parce que la configuration dérive ; un test relancé selon un calendrier montre la dérive tant qu'elle est petite.

> **Remarque :** Une petite équipe avec une base de données et quatre ingénieurs pourrait raisonnablement s'arrêter au troisième palier et consigner ce qui la ferait continuer. La discipline réside dans la raison écrite, pas dans la hauteur atteinte.

La checklist

La checklist est regroupée par pilier, et chaque ligne est une observation. La convention d'usage est la même que pour les tests de sortie : un « non » est un constat avec un responsable et une date, pas un échec à justifier.

```
function runChecklist(items):
    findings = []
    for each item in items:
        answer = observe(item.check)     // read a system, a log or a
table, never ask what is intended
        if answer != "yes":
            findings.add({ item: item.id, owner: nominate(item), due:
dateByWhichToDecide })
    // a finding may end in "accepted, with reason, until a date"; it
may not end in silence
    return findings
```

Autorisation

- ☐ Toutes les tables appartiennent à `ledgerly_owner`, et aucune personne ni aucun service ne détient son login.
- ☐ Chaque service se connecte avec un login membre de son propre rôle `app_*`, et aucun rôle applicatif ne détient de DDL.
- ☐ Aucun accès permanent en écriture n'existe pour une personne ; `human_readonly` est le défaut et `human_jit_write` est un plafond, pas un rôle accordé.
- ☐ Chaque grant JIT et break-glass a un `expires_at`, et le job de révocation comme sa vérification indépendante des grants échus ont tourné récemment.
- ☐ Une connexion directe au primary depuis un ordinateur portable est refusée, et les répliques rejettent le login admin.
- ☐ Les colonnes Restricted n'apparaissent dans aucune demande JIT, aucune session de lecture ni aucune copie d'analytics, et le catalogue de classification liste toutes les colonnes qui existent.
- ☐ Chaque sauvegarde, dump, export et copie de staging a un responsable nommé et une expiration.

Exécution

- ☐ Les instructions des personnes et de l'automatisation atteignent les données à travers la politique d'exécution, et la matrice deny / gate / allow est sous gestion de versions avec un responsable.
- ☐ Les sessions d'écriture s'exécutent avec l'auto-commit désactivé et un row cap, et les guarded settings ne peuvent pas être modifiés depuis l'intérieur d'une session.

- [ ] Une écriture JIT nomme une correction relue, et le runner refuse tout ensemble d'instructions dont le hash diffère de `approved_hash`.
- [ ] Les changements de lien de propriétaire et les écritures dans des colonnes Restricted passent par le chemin produit dans chaque lane.
- [ ] Les changements de schéma s'exécutent en tant que `ledgerly_migrator` depuis le pipeline, et aucune personne ne détient ce login.
- [ ] L'étape d'export s'exécute avec un rôle étroit, et son fichier a un responsable nommé.

Imputabilité

- [ ] Chaque login correspond à exactement une personne ou une instance de workload, et le log de la base de données le montre.
- [ ] Chaque enregistrement d'écriture et de session élevée porte la demande, l'approbateur, le fingerprint et les lignes renvoyées, et les littéraux sont soumis à la redaction.
- [ ] Le stockage d'audit se trouve dans un compte où ni les administrateurs de bases de données ni `ledgerly_access_admin` ne peuvent écrire, et `verifyChain` passe.
- [ ] Un trou dans la séquence, une session de gateway sans session de base de données, ou l'inverse, lève un constat.
- [ ] `evaluateAlertRules` et `runAccessReview` ont tourné lors du dernier cycle, et les révocations qu'ils ont proposées ont été appliquées ou acceptées avec une raison.
- [ ] `containCompromise` a été répété lors d'un game day, et la requête de rayon d'impact a reçu sa réponse à partir de l'enregistrement.

Culture: le chemin sûr doit être le chemin facile

Chaque contrôle du livre est un coût pour la personne qui l'utilise, et cette personne décide, lors de la pire nuit de la semaine, si elle le paie. Un contrôle lent ou souvent en panne est contourné parce que la conception a fait de la route sûre la route difficile, et un incident n'attend pas. La réponse utile consiste à traiter le contournement comme un défaut de la conception.

Je placerais une règle au centre de la culture, et je la rendrais mesurable :

```
function reviewBypass(signal):
    // signal: someone used a second door, a shared login, or an
unplanned break-glass
    cause = ask("what made the intended path unusable at that
moment?")
    if cause in ["slow approval", "gateway unavailable", "missing
product feature", "grant too narrow"]:
        // a design defect: repair the path, and note the date by
which it should be easy
        openFinding(cause, owner: "platform", due: nextReview)
    else if cause == "unknown":
        // an unexplained bypass stays open until it is understood
        keepOpen(signal)
    // nothing here assigns the signal to a person
    return cause
```

Deux habitudes soutiennent la règle. Comptez combien de fois le rôle break-glass est déclaré, et demandez-vous à chaque fois si un chemin JIT ou une fonctionnalité produit aurait dû servir à la place ; un compte qui augmente dit que le chemin normal est trop lent, et un compte nul pendant un an peut signifier que le rôle fonctionne ou que personne ne croit qu'il sera autorisé, ce qu'un game day tranche. Et donnez aux approbateurs quelque chose à quoi ils peuvent répondre la nuit : une approbation qui prend des heures apprend aux demandeurs à demander plus que nécessaire, et une approbation toujours accordée en quelques secondes n'apprend à personne à la lire (le Chapitre 8 couvre la checklist de l'approbateur).

Compromis et coûts

Un chapitre de clôture qui ne ferait que recommander répéterait l'erreur contre laquelle le livre met en garde ; voici donc ce que coûte la conception entière. Le système de l'architecture de référence a beaucoup de pièces, et chacune doit être disponible au moment où quelqu'un en a besoin. La gateway, le runner, le pipeline, le broker, le stockage d'audit et les jobs de revue ont chacun un responsable, un chemin d'astreinte et un mode de défaillance. Une petite équipe plateforme peut découvrir qu'elle passe son temps à faire tourner les contrôles, et c'est un coût réel à mettre en balance avec l'exposition qu'ils suppriment.

La checklist et l'échelle ont leurs propres coûts. Une checklist lue chaque trimestre devient un rituel si les constats ne sont pas clos, et une échelle se transforme en rapport d'avancement quand les équipes optimisent pour le nom du palier ; le test de sortie est l'antidote, parce qu'il est peu coûteux à relancer et difficile à truquer. Une note « fort » dit ce qu'un pattern est construit pour prouver, pas qu'un déploiement donné le prouve.

Quand ne pas utiliser l'appareil de ce chapitre : un système à un seul ingénieur avec une base de données et aucun fournisseur a besoin d'un login propriétaire que personne n'utilise, d'un rôle en lecture seule, d'une sauvegarde qui se restaure et d'un inventaire d'une page. Le livre s'adresse au moment où le nombre de principals ou le coût d'une erreur dépasse cela, et je ne sais pas où se situe ce moment pour votre entreprise ; cela dépend du nombre de personnes qui peuvent atteindre les données, de la part de ces données qui est Restricted, et du temps pendant lequel une erreur reste invisible.

Erreurs courantes

- **Traiter la checklist comme un certificat.** Il est tentant de terminer un passage et d'archiver le résultat. La checklist montre où les preuves manquent le jour où elle a été lue. Relancez-la selon un calendrier, et gardez les constats ouverts jusqu'à ce que chacun soit résolu ou accepté avec une raison et une date.
- **Construire d'abord le pattern le plus à droite.** Un exécuteur médiateur d'instructions séduit parce qu'il a l'air du plus grand contrôle. Sans l'inventaire et des principals distincts en dessous, il sert de médiateur aux instructions d'un login partagé. Gravissez l'échelle dans l'ordre, ou consignez pourquoi un autre ordre vous convient.
- **Ajouter une seconde porte pour les urgences.** Cela ressemble à de la résilience et reconstruit le mot de passe partagé. Si une équipe en a besoin pendant que la gateway mûrit, cette porte a besoin d'un responsable, d'une alerte à l'usage et d'une condition de retrait (Chapitre 8).
- **Lire un contournement comme un problème de conduite.** Quand quelqu'un contourne un contrôle, la conception a rendu cela moins coûteux que le chemin prévu. Demandez ce que coûtait le chemin prévu à ce moment-là, et changez cela.

Points clés

- L'accès aux bases de données en production est un problème d'autorisation, d'exécution et d'imputabilité, et un mot de passe ne touche que le premier, et seulement à la porte.
- Le travail sur les credentials du Chapitre 4 est nécessaire parce que l'identité rend l'enregistrement possible, et il n'est pas suffisant parce qu'il ne dit rien de ce qu'un principal peut faire ni de ce que fait une instruction.
- Choisissez tâche par tâche entre l'accès direct, un bastion, une access gateway, un exécuteur médiateur d'instructions et un chemin produit, avec le pattern le moins coûteux qui prouve encore les piliers que la tâche met en jeu.
- Un palier de maturité est terminé quand son test de sortie passe, et un test que quelqu'un peut relancer est la seule forme de « terminé » qui survit à la dérive.
- Chaque ligne de la checklist est l'observation d'un système, d'un log ou d'une table ; un « non » est un constat avec un responsable et une date, jamais un verdict sur une personne.
- Le chemin sûr doit être le chemin facile, et chaque contournement renseigne sur l'endroit où la conception a rendu la route prévue coûteuse.
- Les contrôles réduisent le rayon d'impact et rendent les comportements visibles, et ils coûtent en disponibilité, en attention et en attente ; nommez ces coûts avant de les adopter, puis à nouveau quand vous les payez.

Alex répond à la question de Robin le lendemain matin par un test et non par une défense : une pièce reste si elle fournit une partie du modèle qu'aucune autre ne fournit et si son test de sortie passe. Le questionnaire reviendra, d'un autre client ou d'un autre auditeur, et les deux mêmes lignes y figureront. Ce que le livre peut vous laisser, c'est un système qui répond à la première ligne à partir d'un document, à la seconde à partir d'un enregistrement, et qui permet à la personne d'astreinte de faire ce qu'il faut sans effort héroïque.

Annexe

Cette annexe est un matériel de référence pour les trois piliers sur lesquels le livre fonde son argument : l'autorisation, l'exécution et l'imputabilité. Elle comporte six parties : un glossaire, une référence rapide des patterns notés au Chapitre 16, la matrice deny / gate / allow du Chapitre 9, une correspondance entre les questions des auditeurs et les artefacts, et deux modèles, l'un pour la revue des accès du Chapitre 14 et l'autre pour la revue post-incident du Chapitre 15.

Rien ici n'est nouveau. Lorsqu'un tableau en reprend un d'un chapitre, les cellules sont celles du chapitre, et si les deux diffèrent, c'est le chapitre qui fait foi. Ce sont vos collègues des services juridique et conformité qui sont responsables de l'interprétation de tout régime ; la correspondance liste des artefacts d'ingénierie et ne prétend rien de plus.

A. Glossaire

Les termes sont classés par ordre alphabétique, avec le chapitre qui construit chacun d'eux. C'est la référence finale du livre pour son vocabulaire : chaque entrée en est la forme courte, et le chapitre nommé en contient l'exposé complet.

Terme	Signification
accès just-in-time (JIT)	Accès accordé pour une tâche précise et une durée limitée (Chapitre 8).
accès permanent	Accès qui existe en continu, qu'il soit utilisé ou non (Chapitre 1).
access gateway	Un intermédiaire conscient de l'identité qui authentifie une personne, émet un credential de courte durée et enregistre la session. Ce n'est ni un bastion ni un proxy de base de données (Chapitre 7).

Terme	Signification
access_audit_record	Les entrées écrites pour chaque instruction, l'une de type intent et l'autre de type outcome, qui contiennent le principal, le login, la demande, le `fingerprint` de l'instruction, les tables et colonnes touchées et, pour l'outcome, le nombre de lignes et le résultat. C'est la partie Enregistrement du modèle (Chapitre 14).
allowlist, denylist	Une liste de ce qui est permis, et une liste de ce qui est refusé. Jamais « whitelist » ni « blacklist » (Chapitre 7).
analyse des requêtes	Inspecter une instruction avant qu'elle ne s'exécute : son type, les tables et colonnes qu'elle touche, si une écriture a un row guard borné, et le nombre de lignes attendu. Elle réduit les erreurs et n'est pas la frontière d'autorisation (Chapitre 9).
approbation liée à l'instruction	Une approbation attachée à l'ensemble exact d'instructions qui s'exécutera, identifié par `approved_hash`, plutôt qu'à une seule fenêtre de temps ou un seul ensemble de privilèges (chapitres 8, 9, 10).
approved hash	L'`approved_hash` d'une demande d'accès : le hash de l'ensemble d'instructions approuvé, pris dans l'enregistrement de merge de la correction relue ou dans l'enregistrement de la demande approuvée, jamais dans l'entrée du runner (chapitres 9, 10).
bastion	Un hôte durci qui constitue le seul chemin réseau vers un système protégé. Il relaie le trafic et ne comprend pas le protocole de la base de données (Chapitre 7).
break-glass	Accès d'urgence avec une journalisation renforcée et une revue obligatoire. Le rôle est `human_breakglass`, et la demande est auto-déclarée (Chapitre 8).
chaîne de hachages	Un format d'enregistrement dans lequel chaque entrée porte le hash de la précédente, si bien qu'une entrée modifiée ou supprimée fait échouer `verifyChain`. Elle fournit une preuve d'altération et ne l'empêche pas (Chapitre 14).
changement de lien de propriétaire	Un `UPDATE` d'une colonne qui relie une ligne au client auquel elle appartient, comme `customer_id` sur `bank_accounts`. Soumis à un gate et dirigé vers le chemin produit (Chapitre 9).
chemin produit	Une opération définie dans un outil, comme `admin-console` demandant à `payments-service` de modifier une colonne Restricted, sans SQL pour la personne (chapitres 9, 10).
classes de données	Les quatre classes nommées de données : **Restricted** (une exposition difficile à réparer, comme `iban` et `token_value`), **Confidential** (données personnelles ou commerciales sur les clients et les contacts), **Internal** (données opérationnelles sans contenu personnel), **Public** (publiables sans risque). Avec une majuscule lorsqu'elles servent de noms de classes (chapitres 2, 12).

Terme	Signification
classification des données	Le tri des données dans les classes nommées qui décident des règles de traitement (Chapitre 12).
confinement	L'arrêt de l'accès d'un principal dans un ordre fixe : révoquer les grants, fermer les sessions, faire la rotation des credentials, délimiter les dégâts, consigner et notifier (Chapitre 15).
copie d'analytics	Le store `ledgerly_analytics`. C'est une copie des données de production, pas une réplique (Chapitre 6).
copie de données	Tout endroit où des données de production ont été dupliquées : répliques, copie d'analytics, sauvegardes, dumps, exports, données de staging (chapitres 6, 13).
correction de données	Une modification des données de production faite par une personne en dehors du chemin applicatif normal, pour réparer un état que l'application ne peut pas réparer (Chapitre 10).
credential	Ce qu'un principal présente pour s'authentifier : un mot de passe, un certificat, un token. Un credential est une sorte de secret (Chapitre 4).
credential de courte durée	Un credential qui expire de lui-même dans un délai court et défini (Chapitre 4).
credential dynamique	Un credential de courte durée généré à la demande pour un seul principal, si bien que deux demandes n'obtiennent jamais la même valeur (Chapitre 4).
credential statique	Un credential qui reste le même jusqu'à ce que quelqu'un le change (Chapitre 4).
deny / gate / allow	Les réponses de la politique d'exécution, toujours dans cet ordre. **Deny** ne s'exécute jamais par ce chemin. **Gate** ne s'exécute qu'après une approbation supplémentaire, ou seulement par le chemin produit. **Allow** s'exécute dans les limites des grants de la session (Chapitre 9).
enregistrement	La partie du modèle qui demande quelle preuve subsiste et si elle relie l'instruction à la personne, à la demande et au résultat (Chapitre 14).
exécution	La façon dont une instruction atteint les données : qui ou quoi la soumet, si quelque chose l'inspecte avant qu'elle ne s'exécute, ce qui borne son effet, et comment elle est validée (chapitres 1, 9).
exécution médiée	Un mode dans lequel le principal ne détient jamais de connexion : il soumet une instruction, la gateway ou un runner l'analyse, et le runner l'exécute (Chapitre 9).
grant	Le fait de donner un privilège à un rôle, un rôle à un login ou à un autre rôle, ou, pour le JIT uniquement, un privilège sur des colonnes nommées directement à un login de session. Désigne aussi l'enregistrement qui en résulte (Chapitre 3).

Terme	Signification
grant au niveau des colonnes	Un grant d'un privilège sur des colonnes nommées d'une table. Là où le moteur ne le permet pas, les colonnes Restricted vivent dans des tables séparées, hors du plafond, et les grants JIT redescendent au niveau de la table (chapitres 3, 8).
guarded settings	Les réglages de session qu'un principal ne peut pas modifier depuis l'intérieur d'une session : timeout d'instruction, row cap, mode d'auto-commit, lecture seule par défaut, rôle. Dits non contournables seulement là où un mécanisme l'impose (chapitres 5, 9).
identité	L'affirmation vérifiée qu'un principal est bien celui qu'il prétend être. Un rôle n'est pas une identité (Chapitre 1).
indépendant du moteur	Le terme du livre pour une notation et des conseils qui n'appartiennent à aucun moteur de base de données (Préface).
log d'audit	L'enregistrement de qui a fait quoi sur quelles données, conservé là où l'acteur ne peut pas le modifier. La table `audit_events` est l'enregistrement métier de l'application, pas le log d'audit (Chapitre 14).
login	Le compte au niveau de la base de données avec lequel un principal se connecte (chapitres 1, 3).
masquage	Le remplacement ou l'occultation de valeurs, de sorte que la vraie valeur ne soit pas visible tandis que la forme reste utilisable. Le masquage statique modifie une copie, le masquage dynamique modifie ce que renvoie une requête (Chapitre 12).
migration	Un changement de schéma versionné exécuté par `ledgerly_migrator`. « Changement de schéma » est l'idée générale, « migration » l'artefact (Chapitre 11).
minimisation	Ne pas collecter, copier ni conserver des données dont personne n'a besoin (Chapitre 12).
moindre privilège	N'accorder que ce dont une tâche a besoin, pour la durée dont elle a besoin, en comptant le coût de cette démarche (chapitres 1, 3).
PII	Informations personnelles identifiables. « Données sensibles » est le terme générique pour les données Restricted et Confidential (Chapitre 2).
plafond	Le maximum qu'une demande JIT peut solliciter, défini par `human_jit_write`. Jamais une colonne Restricted (chapitres 3, 8).
porte latérale	Une copie de données ou un chemin d'export qui contourne les contrôles d'accès du primary (chapitres 6, 13).
primary, réplique	La base de données accessible en écriture et ses copies en lecture. La copie d'analytics n'est pas une réplique (Chapitre 6).
principal	Tout ce qui s'authentifie : une personne, un service, un job, une intégration de fournisseur (Chapitre 1).

Terme	Signification
principal d'automatisation	Un principal qui n'est pas une personne : un job, un runbook, une commande ChatOps, un agent d'IA. Il a une workload identity et la lecture seule par défaut, et l'enregistrement nomme à la fois ce principal et la personne qui l'a déclenché (Chapitre 9).
principe des quatre yeux	Une approbation dans laquelle une seconde personne relit avant qu'une action risquée ne s'exécute. Il dit qui approuve, pas ce qui est approuvé (Chapitre 8).
privilège	Une chose précise qu'un grant permet, comme `SELECT` sur une table (Chapitre 1).
proxy de base de données	Un intermédiaire qui comprend le protocole et qui mutualise, route ou filtre les connexions à la base de données. On l'appelle pooler quand il n'est question que de pooling (chapitres 5, 7).
rayon d'impact	Ce qui peut être affecté si un principal, un credential ou une action tourne mal. Toujours énoncé à propos d'une chose précise (chapitres 1, 2).
redaction	La suppression complète d'une valeur dans une sortie ou un log (Chapitre 12).
requête ad hoc	Une requête qu'une personne écrit et exécute une fois, en dehors de toute application (Chapitre 10).
révocation, rotation	La révocation fait cesser immédiatement le fonctionnement d'un credential ou d'un grant. La rotation remplace un credential selon un calendrier ou sur soupçon, et ne ferme pas les sessions déjà ouvertes (chapitres 4, 15).
revue des accès	Une vérification planifiée que l'accès de chaque principal est encore nécessaire. Elle compare ce qui est accordé avec ce qui a été utilisé, et un relecteur nommé décide (Chapitre 14).
rôle	Un ensemble nommé de privilèges dans la base de données. Un rôle n'est jamais une personne (Chapitre 3).
runbook	Une procédure écrite et suivie pour une tâche répétable (Chapitre 10).
runner	Le runner de corrections de données relu. Il n'exécute un ensemble d'instructions que si son hash est égal à `approved_hash`, et la gateway ne relaie les écritures JIT que depuis lui (Chapitre 10).
secret	Toute valeur qui donne un accès si elle est divulguée (Chapitre 4).
session	Une connexion authentifiée ou une interaction relayée par la gateway, pour un principal (chapitres 5, 7).
session handout	Un mode dans lequel la gateway donne à un principal un login de session et une connexion, et c'est le client propre du principal qui envoie les instructions. Les grants du login bornent tout (Chapitre 9).
théâtre de sécurité	Un contrôle qui ajoute de la friction sans réduire l'exposition (chapitres 2, 8).

Terme	Signification
tokenisation	Le remplacement d'une valeur par un substitut que seul un service contrôlé peut ramener à l'original. Ce n'est ni du chiffrement ni du masquage (Chapitre 12).
workload identity	Une identité que la plateforme donne à un workload en cours d'exécution pour qu'il puisse s'authentifier sans mot de passe stocké (Chapitre 4).

Plusieurs de ces paires sont souvent confondues, et une règle courte tranche chacune avant qu'une discussion ne commence.

Si la question porte sur	Employer	Et non
Un hôte qui relaie le trafic réseau, sans rien savoir des instructions	bastion	access gateway
Un intermédiaire qui émet des logins de session par personne et enregistre les sessions	access gateway	proxy de base de données
La mutualisation ou le routage des connexions, rien sur les personnes	pooler, ou proxy de base de données	access gateway
Le client propre de la personne envoie des instructions sur une connexion que la gateway lui a donnée	session handout	exécution médiée
La personne soumet une instruction et ne détient jamais de connexion	exécution médiée	session handout
Faire cesser immédiatement le fonctionnement d'un credential	révocation	rotation
Qui approuve une action risquée	principe des quatre yeux	approbation liée à l'instruction
Ce qui a été approuvé exactement, par hash	approbation liée à l'instruction	principe des quatre yeux
L'enregistrement métier de l'application	`audit_events`	log d'audit

B. Référence rapide des patterns

Le Chapitre 16 note cinq patterns au regard des trois piliers. Les notes relèvent du jugement du livre et non d'une mesure : « fort » signifie que le pattern est construit pour prouver le pilier, « partiel » qu'il en prouve une partie ou seulement si l'on ajoute autre chose, « aucun » qu'il ne dit rien. La colonne des coûts est celle que donne le Chapitre 16, raccourcie.

Pattern	Ce qu'il résout	Coût	Autorisation	Exécution	Imputabilité	Chapitres
Accès direct (du client à la base de données)	Un login existe et ses grants étaient ce qu'ils étaient	Nul à construire ; tous les coûts arrivent plus tard, sous la forme d'une semaine de fouilles	Partiel : grants seulement, la portée est ce que le réseau permet	Aucun	Partiel au mieux : des logins par personne si quelqu'un les provisionne, sinon aucun	1
Bastion	Un hôte contrôlé unique, un seul endroit où couper le chemin réseau	Un hôte à patcher et à maintenir disponible	Partiel : rétrécit le chemin réseau	Aucun : il relaie, il ne distingue pas un `SELECT` d'un `DROP`	Partiel : qui s'est connecté à l'hôte, pas ce qui s'est exécuté	7
Session proxy ou access gateway	Une personne nommée a détenu un login de session précis, entre deux horodatages, sous des grants précis	Un nouveau système qui doit être disponible quand quelqu'un en a besoin, et le chemin d'approbation qui l'accompagne	Fort : logins par session, expiration, grants par demande	Partiel : borné par les grants et les guarded settings, les instructions sont tapées librement	Fort sur le qui et le quand, partiel sur le quoi	7, 8
Exécuteur médiateur d'instructions	Cet ensemble exact d'instructions, par son hash, a été approuvé, inspecté, borné et compté	Chaque tâche attend la revue, rien n'est interactif, quelqu'un construit et maintient le runner	Fort	Fort : inspecté avant de s'exécuter, lié à un approved hash	Fort : instruction, approbateur et résultat dans un seul enregistrement	9, 10
Chemin produit (sans SQL)	Une opération définie s'est exécutée, pour un rôle dans un outil	Une fonctionnalité par tâche, et le cycle de changement le plus lent des cinq	Fort : l'opération est la permission	Fort : l'opération est fixe	Partiel : la personne figure dans la piste d'audit de l'outil, le log de la base de données montre le login du service	9, 10

Choisir parmi eux est une décision prise tâche par tâche, et la règle est celle que donne le Chapitre 16 : le pattern le moins coûteux qui prouve encore les piliers que la tâche met en jeu.

```
function choosePattern(task):
    // a person who never opens a session cannot leave one open
    if task.repeats and task.hasFixedShape:
        return "product path"
    // a schema change takes the migration pipeline, never the runner
(Chapter 11)
    if task.changesSchema:
        return "migration pipeline"
    // Restricted columns and owner links are changed by the product,
not by SQL
    if task.writesRestrictedColumn or task.changesOwnerLink:
        return "product path"
    if task.writesData:
        // reviewed statement set, approved hash, runner executes
        return "statement-mediating executor"
    if task.readsData:
        return "access gateway: human_readonly on ledgerly_replica_2"
    // anything else is an exception with a named owner and an end
    return "exception: owner, reason, ends_on"
```

C. Matrice des opérations dangereuses

La matrice ci-dessous est la politique d'exécution du Chapitre 9, reprise avec ses cellules inchangées. Les lignes sont les classes d'instructions que produit le classificateur. Les colonnes sont les lanes par lesquelles une instruction peut arriver. Chaque cellule porte l'une de trois réponses : deny (ne s'exécute jamais par ce chemin), gate (a besoin d'une approbation qu'elle n'a pas encore, ou relève d'une autre route), allow (s'exécute dans les limites de la boucle et, là où une cellule dit « borné », seulement à l'intérieur de la borne fixée par la source nommée).

Classe d'instruction	Session de lecture humaine	Écriture JIT	Break-glass	Runner de corrections de données	Migration
Lecture, dans le row cap	Allow	Allow	Allow	Allow	Allow
Écriture bornée, dans le décompte approuvé	Deny	Allow, borné à la correction (runner seulement)	Allow, borné par un row cap (note obligatoire)	Allow, borné à l'approved hash et au décompte	Allow, borné par la règle de backfill

Classe d'instruction	Session de lecture humaine	Écriture JIT	Break-glass	Runner de corrections de données	Migration
Écriture non bornée	Deny	Deny	Deny	Deny	Gate : approbateur nommé
Changement de lien de propriétaire	Deny	Gate : chemin produit	Gate : chemin produit	Gate : chemin produit	Gate : approbateur nommé
Écriture dans une colonne Restricted	Deny	Gate : chemin produit	Gate : chemin produit	Gate : chemin produit	Gate : approbateur nommé
Export	Gate : selon le palier	Deny	Deny	Deny	Gate : approbateur nommé
Changement de schéma (DDL)	Deny	Deny	Deny	Deny	Allow : lane relue
Changement de privilèges	Deny	Deny	Deny	Deny	Gate : revue plateforme
Écriture dans `ledger_entries` ou `audit_events`	Deny	Deny	Deny	Deny	Gate : revue plateforme

Trois règles de lecture évitent la plupart des recherches.

- **Une écriture entre les mains d'une personne est refusée ou bornée.** La session de lecture humaine n'a pas de chemin d'écriture, les écritures JIT ne viennent que du runner, et le break-glass est la seule lane où une personne écrit directement, sous un row cap et avec une note.
- **Un gate nomme l'endroit où l'instruction va ensuite.** « Chemin produit » signifie que `admin-console` demande à `payments-service`, qu'une seconde personne approuve et que le changement est enregis-

tré dans `audit_events`. « Approbateur nommé » et « revue plateforme » signifient que la lane de migration signale l'instruction à une personne qui doit dire oui.

- **La matrice n'est pas la frontière d'autorisation.** L'analyse des requêtes réduit les erreurs. Les grants du login restent la frontière, et une instruction que le classificateur lit mal reste limitée par eux (Chapitre 9).

La décision pour une instruction, écrite comme une règle :

```
function answerFor(statementClass, lane):
    // the matrix above is the only source; no lane has its own rule
set
    cell = matrix[statementClass][lane]
    if cell starts with "Deny":
        refuse and record the refusal
    else if cell starts with "Gate":
        // the text after the colon names the route or the approver
        hold the statement; route it as the cell says; do not run it
as sent
    else:
        // "bound" cells compare the real count with the bound before
COMMIT
        run within the loop's limits
```

D. Correspondance avec les exigences de conformité

Les régimes de protection des données et des cartes de paiement posent en général une même famille de questions sur les accès. Un auditeur qui travaille dans le cadre de l'un de ces régimes voudra le plus souvent voir qu'un contrôle existe, qu'il a tourné et que ses résultats ont été examinés. Cette correspondance liste les types de questions, le contrôle qui répond à chacune, l'artefact qu'il laisse et le chapitre qui le construit. C'est le tableau du Chapitre 14, repris, et il ne prétend pas qu'un contrôle satisfait une quelconque obligation.

Question	Contrôle	Artefact	Chapitre
Qui peut atteindre les données de production, et par quel chemin ?	La gateway comme seule porte, le schéma de rôles en tant que code, la vérification de dérive	Liste des règles réseau, définitions des rôles, rapport de dérive	3, 7

Question	Contrôle	Artefact	Chapitre
Qui a exécuté une instruction donnée, avec quels grants et quelle approbation ?	`access_audit_record` avec l'id de la demande et l'approbateur	L'enregistrement et le rapport de `verifyChain`	14
Combien de temps dure un accès élevé, et qui l'a approuvé ?	Demande JIT avec approbateur et `expires_at`, job de révocation	Enregistrements des demandes, log de révocation, vérification des grants en dépassement	8
Comment les données sensibles sont-elles identifiées et limitées ?	Classification dans le schéma, masquage, colonnes retenues	Le catalogue, la vérification du chargement masqué	12
Où se trouvent les copies des données de production, et qui est responsable de chacune ?	Registre avec responsable et date de fin, sweeper	Registre, log du sweeper	6, 13
Qui modifie le schéma, et qui l'a relu ?	Pipeline de migration, branche protégée, entrée de démarrage externe	Enregistrements de migration rapprochés des commits et des entrées d'audit	11
L'accès est-il encore nécessaire ?	Revue des accès avec des relecteurs nommés	Revue terminée avec ses décisions	14
Que se passe-t-il en cas d'urgence ?	Break-glass avec alerte et revue obligatoire	Enregistrements des revues, historique des alertes	8, 15
Qui administre l'accès lui-même ?	Logins `ledgerly_access_admin` en tant que producteurs séparés	Leurs flux et la comparaison entre catalogue et enregistrement	3, 14

Un contrôle qui existe mais n'a jamais tourné, ou qui a tourné sans que personne n'en examine les résultats, ne répond qu'à un tiers de la question. La vérification ci-dessous est celle à exécuter avant de remettre un artefact à quiconque l'a demandé.

```
function evidenceStatus(row):
    // three things an auditor tends to want, from Chapter 14
    if row.artifact is null:
        return "control not evidenced: build or find the artifact"
    if row.artifact.lastProducedAt is null:
        return "control exists, no evidence it ran"
    if row.artifact.lastReviewedBy is null:
        return "control ran, results not looked at"
    return "evidenced: attach artifact and its review"
```

E. Modèle de revue des accès

Voici le modèle qui se trouve derrière le `runAccessReview` du Chapitre 14. Copiez-le pour chaque cycle de revue. La revue propose et un relecteur nommé décide ; personne ne revoit son propre accès, et la période est plus longue que le plus long cycle du workload revu (une revue sur un mois proposerait de révoquer un grant qu'un rapport trimestriel utilise deux mois sur trois).

En-tête

```
Review cycle:            <period start> to <period end>
Reviewer(s):             <names; none is the owner of an item they
review>
Record queried on:       <date>, verifyChain result: <passed | failed:
details>
Period chosen and why:   <longest workload cycle covered?>
```

Partie 1 : grants déclarés face à l'usage

Grant (rôle, privilège, table) ou login	Respon-sable	Dernier usage dans l'enre-gistre-ment	Constat	Décision	Décidé par	Fait le
`<role>`, `<privilege>`, `<table>`	`<named principal>`	`<date or none>`	inutilisé / sans respon-sable / usage plus étroit que le grant / correct	conserver / re-streindre à `<privilege>` / ré-voquer / désigner un responsable	`<reviewer>`	`<date>`

Rapprochez l'usage par privilège, et pas seulement par table. Un rôle qui fait des sélections chaque nuit et n'a jamais fait de mise à jour est candidat à une restriction de `S, U` à `S`. Incluez les correspondances entre groupes

du fournisseur d'identité et rôles, ainsi que les logins de `ledgerly_access_admin`.

Partie 2 : ce qui n'est pas un grant

Élément	Question	Réponse	Responsable	Décision
Registre des copies de données (Chapitre 13)	Chaque copie a-t-elle encore un responsable et une date de fin future ?	`<yes / no, which>`	`<name>`	`<keep / end by date>`
Qui peut approuver les demandes JIT	Chaque approbateur occupe-t-il encore un rôle qui le justifie ?	`<yes / no, which>`	`<name>`	`<keep / remove>`
Qui peut lire le stockage d'audit	Chaque lecteur en a-t-il encore besoin ?	`<yes / no, which>`	`<name>`	`<keep / remove>`
Principals d'automatisation (jobs, runbooks, agents d'IA)	Chacun tourne-t-il encore, avec une workload identity et la lecture seule par défaut ?	`<yes / no, which>`	`<name>`	`<keep / retire>`

Partie 3 : usages du break-glass sur la période

Id de la demande	Utilisé par	Incident ou raison	Revu par	Date de la revue	Constat sur le système
`<id>`	`<principal>`	`<reason>`	`<other than user>`	`<date or MISSING>`	`<for example: missing path>`

Une ligne sans date de revue est en soi un constat.

Règles de validation

```
function closeAccessReview(review):
    for each row in review.rows:
        // a row without a decision is not a review, it is a form
        if row.decision is null or row.decidedBy is null:
            raise ReviewIncomplete
        if row.decidedBy == row.owner:
            raise ReviewerIsOwner
    // decisions become tracked work with an owner and an end date
    for each row in review.rows where row.decision != "keep":
        createChange(row, owner: row.owner, endsOn:
review.nextCycleStart)
    markClosed(review.id)
```

F. Modèle de revue post-incident

Ce modèle suit le Chapitre 15. Il est sans blâme : chaque question porte sur le système qui a permis une action ou l'a rendue invisible, et aucune ne demande qui aurait dû faire preuve de plus de discernement. Remplissez-le à partir de l'enregistrement, pas de mémoire. Lorsque l'enregistrement ne peut pas répondre à une question, écrivez-le, parce que le manque est un constat. Les heures sont celles que montre l'enregistrement ; ne les estimez pas.

1. Résumé

```
Incident id:         <id carried in every request reason>
Opened / closed:     <timestamps from the incident record>
Trigger:             <alert, report or declared exposure>
Principal(s) in scope: <logins, services, automation principals>
Pillars touched:     <authorization / execution / accountability>
```

2. Confinement, dans l'ordre

Étape	Faite à	Issue	Remarques sur ce qui a gêné
Révoquer les grants	`<timestamp>`	fait / échoué	
Fermer les sessions	`<timestamp>`	fait / échoué	
Faire la rotation des credentials	`<timestamp>`	fait / échoué	
Délimiter les dégâts à partir du log d'audit	`<timestamp>`	fait / échoué	
Consigner et notifier	`<timestamp>`	fait / échoué	

Questions : chaque étape s'est-elle exécutée même si une étape précédente avait échoué ? Des étapes sont-elles restées inachevées, et ont-elles été suivies ? Le log était-il joignable et, sinon, les entrées ont-elles été mises en file d'attente sans retarder le retrait des accès ?

3. Rayon d'impact

```
Query run:           <everything the principal touched since T,
Restricted rows per table>
Tables touched:      <list, with class>
Restricted columns:  <touched or not, rows returned per table>
Agreement check:     <does the scope agree with the register of copies
and the gateway session list?>
```

4. Cycle de vie des accès pendant l'incident

Id de la demande	Type	Demandé	Accordé	Premier usage	Révoqué	Revu par
`<id>`	jit / breakglass	`<time>`	`<time>`	`<time or none>`	`<time>`	`<other than user, or MISSING>`

Questions : un grant est-il resté au-delà de `expires_at` ? Un accès a-t-il été accordé sans qu'aucune demande ne l'explique ? `closeIncident` a-t-il trouvé des accès ouverts, et ont-ils été fermés avant l'incident ?

5. Écritures pendant l'incident

Écriture	Route (runner avec revue accélérée, chemin produit, break-glass brut)	Relecteur qui n'était pas l'auteur	Approved hash concordant
`<description>`	`<route>`	`<name>`	oui / non

Question : le break-glass brut a-t-il servi pour une écriture alors que le runner aurait suffi, et pourquoi ?

6. Cas différés, s'ils se sont produits

- **Gateway indisponible :** combien de temps l'accès humain a été interrompu, si les services ont continué de tourner, et si le runbook de rétablissement a fonctionné tel qu'il est écrit.
- **Restauration pendant l'incident :** si la copie restaurée a été inscrite au registre avec un responsable et une date de fin, et si elle a pris fin.
- **Approbations lentes la nuit :** quelle demande a attendu, pour qui, et ce qu'a fait la règle prévue pour une demande sans réponse.

7. Constats et suivi

Constat sur le système	Partie (Identité, Grant, Chemin, Exécution, Enregistrement)	Change-ment	Respon-sable	Fin le
`<for example: a path that should have existed>`	`<part>`	`<change>`	`<name>`	`<date>`

Terminez par une vérification : chaque constat est-il un changement apporté à une conception, avec un responsable et une date de fin, et non un rappel à plus de prudence ?

THE UNRELIABLE
SYSTEMS TRILOGY

I

À PROPOS DU LIVRE

L'accès aux bases de données en production n'est pas un problème de distribution de credentials ; c'est un problème d'autorisation, d'exécution et d'imputabilité.

Un meilleur coffre-fort de mots de passe ne change que la façon dont un secret parvient à une personne. Il ne décide ni de ce que cette personne peut faire, ni de ce qui s'exécute réellement sur les données, ni de qui en répond ensuite. Ce livre suit Ledgerly, d'un mot de passe admin partagé à un modèle d'accès conçu délibérément : identités à moindre privilège, accès just-in-time et break-glass, exécution médiée pour les opérations dangereuses, et une piste d'audit capable de dire qui a fait quoi, et pourquoi. Les patterns sont indépendants du moteur, et chacun est présenté avec le coût qu'il comporte.

À PROPOS DE L'AUTEUR

Muhammet Şafak développe des logiciels depuis 2008. Pendant de longues années, Muhammet a occupé des rôles actifs dans les domaines de la santé, de la finance et des télécommunications, et défend l'idée que le développement doit partir du problème, non de la technologie. Depuis 2024, Muhammet conçoit et développe des systèmes à fort trafic capables de monter en charge de bout en bout.

Muhammet accompagne en mentorat des équipes de développement logiciel de petite et moyenne taille, mène des projets open source et consigne ses décisions par écrit, avec leurs raisons et leur coût.

Les systèmes fiables ne se construisent pas avec des composants fiables. Ils se construisent autour des composants qui défaillent.

I ACCÈS

II ÉVÉNEMENTS

III AGENTS