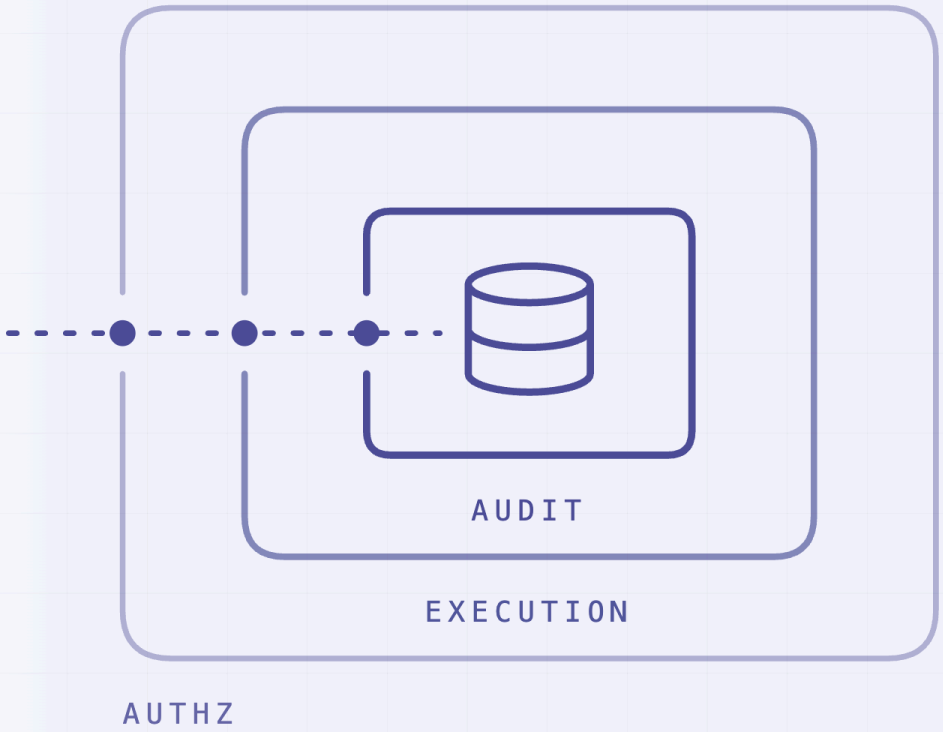
THE UNRELIABLE
SYSTEMS TRILOGY



Leitfaden für den Zugriff auf Produktionsdatenban

Muhammet Şafak

Leitfaden für den Zugriff auf Produktionsdatenbanken

Identitäten, Credentials, Just-in-Time-Zugriff, vermittelte Ausführung und Audit

Muhammet Şafak

© 2026 Muhammet Şafak
Kostenlos herunterladen und lesen.

Inhalt

Vorwort 6

TEIL I — GRUNDLAGEN **14**

Kapitel 1 — Die Tür, die niemand entworfen hat 15

Kapitel 2 — Ein Bedrohungsmodell für Produktionsdaten 31

Kapitel 3 — Identitäten, Rollen und minimale Rechte 51

TEIL II — ZUGRIFF DURCH ANWENDUNGEN **76**

Kapitel 4 — Credentials und Secrets 77

Kapitel 5 — Verbindungen als Richtlinie 98

Kapitel 6 — Lesezugriffe, Replikate und Datenkopien 118

TEIL III — ZUGRIFF DURCH MENSCHEN **133**

Kapitel 7 — Zugangspfade: Bastionen, Proxies und Access-Gateways 134

Kapitel 8 — Just-in-Time- und Break-Glass-Zugriff 158

Kapitel 9 — Vermittelte Ausführung: Abfrageanalyse und gefährliche Operationen 185

Kapitel 10 — Ad-hoc-Abfragen und Datenkorrekturen 203

Kapitel 11 — Schemaänderungen und Zugriff für Migrationen 230

TEIL IV — DATEN SCHÜTZEN **253**

Kapitel 12 — Sensible Daten: Klassifizierung, Maskierung und Minimierung 254

Kapitel 13 — Backups, Exporte und andere Seitentüren 278

TEIL V — NACHWEIS UND BETRIEB **303**

Kapitel 14 — Auditierung und Zurechenbarkeit 304

Kapitel 15 — Zugriff bei Incidents 325

Kapitel 16 — Eine Checkliste für den Zugriff und Schlussgedanken 345

Anhang 361

Vorwort

Stellen Sie sich einen Donnerstagabend bei Ledgerly vor, einem Unternehmen für Rechnungsstellung und Zahlungen, das es nicht gibt. Robin hat Rufbereitschaft. Die Rechnung eines Kunden hängt seit dem Nachmittag im Zustand „sending“, und die Korrektur sieht klein aus: eine Zeile, eine Statusspalte. Der Desktop-Datenbank-Client des Teams merkt sich das gemeinsame Admin-Passwort, verbindet sich mit dem Primary und führt automatisch einen Commit aus, wie es solche Clients oft tun. Die Korrektur einer einzigen Zeile läuft über diesen Client, unter einer Rolle, die jede Tabelle ändern darf, und die Klausel `WHERE` der Anweisung trifft statt einer Rechnung gleich einen ganzen Batch von Rechnungen.

Robin verbringt den Rest des Abends mit der Reparatur. Länger dauert die Frage, die danach kommt: Wer hat diese Anweisung ausgeführt? Das Datenbank-Log zeigt die gemeinsame Admin-Rolle, die jeder Service und, sagen wir, zwanzig Personen nutzen. Der Passwortmanager zeigt, wer den Eintrag geöffnet hat, nicht wer was ausgeführt hat. Das Team kann die Änderung anhand der Seiteneffekte auf Seiten der Anwendung rekonstruieren, und schließlich gelingt ihm das auch. Aber etwa einen Tag lang kann niemand sicher sagen, was genau passiert ist, welche Zeilen betroffen waren und ob zur selben Zeit noch etwas anderes dieselbe Tür benutzt hat.

Nichts in dieser Geschichte war für sich genommen unvernünftig. Das Passwort wurde geteilt, weil das der schnellste Weg war, den ersten Service zum Laufen zu bringen. Der Client führte automatisch Commits aus, weil das seine Voreinstellung ist. Die Rolle durfte jede Tabelle ändern, weil es anfangs einfacher war, als zu entscheiden, welche Tabellen jeder Service braucht. Jeder Schritt war an seinem Tag vernünftig. Zusammen bauten sie eine Tür, die niemand entworfen hat.

Diese Tür ist das Thema dieses Buches.

Warum es dieses Buch gibt

Der Zugriff auf eine Produktionsdatenbank wird meist wie Installationstechnik behandelt. Er wird eingerichtet, wenn der erste Service ausgerollt wird, und danach nur noch angefasst, wenn etwas kaputtgeht oder jemand das Unternehmen verlässt. Währenddessen wird die Datenbank zum wertvollsten Bestandteil dessen, was das Unternehmen betreibt: die Kunden, das Geld, die Geschichte. Die Lücke zwischen der Abhängigkeit des Unternehmens von der Datenbank und dem geringen Entwurfsaufwand für ihren Zugriff ist der Gegenstand dieses Buches.

Über jeden einzelnen Baustein ist schon viel geschrieben worden. Die Datenbankdokumentation erklärt, wie man Rollen und Grants anlegt. Tools für Secrets erklären, wie man Credentials ausstellt. Sicherheitsleitfäden erklären, warum Dauerzugriff ein Risiko ist. Schwerer zu finden ist das Gesamtbild aus der Sicht von Entwicklern: wie die Bausteine zusammenpassen, was jeder von ihnen die Menschen kostet, die das System bauen und betreiben, und in welcher Reihenfolge ein Team mit knapper Zeit sie einführen könnte.

Dieses Buch versucht, dieses Bild zu sein, und es geht von einer einzigen Behauptung aus. Der Zugriff auf Produktionsdatenbanken ist kein Problem der Credential-Verteilung; es ist ein Problem der Autorisierung, der Ausführung und der Zurechenbarkeit.

Die gängige Sichtweise behandelt Zugriff als Frage, wer ein Passwort besitzt und wie es gespeichert, rotiert und weitergegeben wird. Der Donnerstagabend oben zeigt, wo diese Sichtweise endet. Robin besaß ein legitimes Credential, und das Credential tat genau das, wofür es ausgestellt war. Versagt hat alles drumherum: Nichts entschied, dass diese Person eine Schreiboperation auf dieser Tabelle ausführen darf, nichts prüfte die Anweisung oder begrenzte ihre Wirkung, bevor sie mit einem Commit abgeschlossen wurde, und nichts ordnete das Ergebnis hinterher einer Person zu. Die Autorisierung entscheidet, was geschehen darf und wo. Die Ausführung ist der Weg, auf dem jede Anweisung die Daten erreicht. Die Zurechenbarkeit ist der Nachweis, der übrig bleibt. Jedes Kapitel nennt die Säule, der es dient, und Teil I baut das Vokabular für alle drei auf. Das Buch behandelt jede Kontrolle als Entscheidung mit einem Preis. Eine

Kontrolle, die für die Tabelle mit den Zahlungs-Tokens richtig ist, kann für eine Tabelle mit Rechnungsvorlagen sinnlos sein, und das Buch sagt es, wenn es darauf ankommt.

Ich sollte meine Haltung klar benennen. Das Buch ist aus der Sicht der Verteidiger geschrieben. Bedrohungen werden nur so weit beschrieben, wie sie erklären, warum eine Kontrolle existiert; nichts hier ist eine Anleitung zum Eindringen. Es verkauft keine Angst und zitiert keine Statistiken zu Sicherheitsvorfällen, denn ein ruhiges Design-Review ist nützlicher als ein alarmierendes. Und es rechnet die Reibung ehrlich mit. Jede Kontrolle verlangt etwas von den Entwicklern und von der Person in Rufbereitschaft, und eine Kontrolle, die Reibung erzeugt, ohne die Angriffsfläche zu verkleinern, ist Sicherheitstheater. Wenn ich eine verbreitete Praxis für Theater halte, sage ich es und erkläre, warum.

Für wen dieses Buch ist

Das Buch ist geschrieben für **Backend- und Plattform-Entwickler**, die Services bauen, die mit einer Produktionsdatenbank sprechen, für **SRE- und DBA-Rollen**, die diese Datenbank betreiben, für **Tech Leads**, die entscheiden müssen, wie Zugriff funktionieren soll, und für **Entwickler, die neu für Sicherheit zuständig geworden sind**, in einem Team, das nie eine eigene Sicherheitsfunktion hatte. Wenn Sie sich schon einmal gefragt haben, ob das gemeinsame Passwort noch das gemeinsame Passwort ist, gezögert haben, bevor Sie einem Kollegen Schreibzugriff gaben, oder gefragt wurden: „Wer kann diese Tabelle sehen?“ und gemerkt haben, dass die ehrliche Antwort einen Tag Nachforschung braucht, dann haben Sie genau die Leserschaft vor Augen, für die ich schreibe.

Sie brauchen keinen Sicherheitshintergrund. Mit einigen Dingen sollten Sie vertraut sein:

- Relationale Datenbanken: Tabellen, Rollen und Grants, Transaktionen und was ein Replikat ist.
- Services, die sich über ein Netzwerk mit einer Datenbank verbinden, und das Vokabular rund um Connection-Pools und Timeouts.
- Arbeit in Rufbereitschaft: alarmiert werden, unter Druck etwas reparieren und danach aufschreiben, was geschehen ist.

Vertrautheit mit einer bestimmten Datenbank-Engine hilft bei den Beispielen, ist aber nicht erforderlich, wie ein späterer Abschnitt erklärt.

Was dieses Buch nicht ist

Es ist kein Buch über Datenbankadministration. Es bringt Ihnen nicht bei, einen Planner zu tunen, einen Buffer zu dimensionieren oder eine Replikationstopologie zu wählen. Wo Administration für den Zugriff eine Rolle spielt, etwa bei Replikaten und Migrationen, behandelt das Buch diesen Teil und überlässt den Rest der Dokumentation Ihrer Datenbank.

Es ist kein Handbuch zur Compliance. Regelwerke wie DSGVO, KVKK, PCI DSS, SOC 2 und HIPAA kommen vor, weil sie prägen, was Auditoren und Kunden fragen, und das Buch beschreibt sie nur in diesem Sinn: Regelwerke dieser Art fragen tendenziell, wer was sehen kann, wie Zugriff genehmigt wird und welche Aufzeichnung danach existiert. Es legt kein Regelwerk aus, zitiert keine Artikel und sagt Ihnen nicht, was von Ihnen verlangt wird. Diese Auslegung liegt bei Ihrem Team aus Recht und Compliance, und das Ziel des Buches ist, ihnen eine Architektur zu geben, über die sie urteilen können.

Und es ist kein Angriffsbuch. Sie finden hier keine Techniken zum Einbrechen. Wenn ein Kapitel eine Bedrohung braucht, benennt es das Risiko in einem Satz, etwa ein Credential, das offengelegt wird, eine Abfrage, die auf der falschen Datenbank landet, oder eine offen gelassene Vendor-Session, und geht weiter zur Kontrolle. Wenn eine genauere Beschreibung der Bedrohung das Design nicht ändern würde, bleibt das Detail weg.

Wie das Buch aufgebaut ist

Das Buch hat fünf Teile und sechzehn Kapitel, gefolgt von einem Anhang.

Teil I, Grundlagen (Kapitel 1 bis 3) begründet die These oben und baut das Vokabular für den Rest auf. Kapitel 1 betrachtet, wie sich Türen zu einer Datenbank ansammeln. Kapitel 2 baut ein Bedrohungsmodell für Produktionsdaten auf, das neben Missbrauch auch Fehler einschließt. Kapitel 3 behandelt Identitäten, Rollen und minimale Rechte, von denen jedes spätere Kapitel abhängt.

Teil II, Zugriff durch Anwendungen (Kapitel 4 bis 6) handelt von den Services. Kapitel 4 behandelt Credentials und Secrets, von statischen Passwörtern bis zu dynamischen Credentials und Workload Identity. Kapitel 5 behandelt Verbindungen als Richtlinie: Pooling, Limits und Timeouts. Kapitel 6 behandelt Lesezugriffe, Replikate und die anderen Kopien der Daten, die die Produktion unbemerkt erzeugt.

Teil III, Zugriff durch Menschen (Kapitel 7 bis 11) handelt von Menschen. Kapitel 7 vergleicht Bastionen, Proxies und Access-Gateways. Kapitel 8 behandelt Just-in-Time- und Break-Glass-Zugriff. Kapitel 9 behandelt die Ausführung selbst: wie ein Gateway jede Anweisung prüft, ihre Wirkung begrenzt und sie mit einem Commit abschließt. Kapitel 10 behandelt Ad-hoc-Abfragen und Datenkorrekturen. Kapitel 11 behandelt Schemaänderungen und wer sie vornehmen darf.

Teil IV, Daten schützen (Kapitel 12 und 13) verlagert den Blick von der Frage, wer sich verbinden kann, auf die Frage, was sich einsehen lässt. Kapitel 12 behandelt Klassifizierung, Maskierung und Minimierung. Kapitel 13 behandelt Backups, Exporte und andere Seitentüren, die alle bisher gebauten Kontrollen umgehen.

Teil V, Nachweis und Betrieb (Kapitel 14 bis 16) behandelt, woran ein Team erkennt, dass seine Kontrollen funktionieren, und wie es sie funktionsfähig hält. Kapitel 14 behandelt Audit-Logs und Zugriffs-Reviews. Kapitel 15 behandelt den Zugriff bei Incidents, wenn der Druck am höchsten und die Abkürzungen am verlockendsten sind. Kapitel 16 bietet eine Checkliste, eine Reifegradleiter mit Ausstiegstests und einige abschließende Gedanken zur Kultur.

Der Anhang versammelt ein Glossar, eine Kurzreferenz der Patterns, eine Matrix gefährlicher Operationen, eine Compliance-Zuordnung sowie Vorlagen für Zugriffs-Reviews und Post-Incident-Reviews.

Jedes Kapitel hat dieselbe Gestalt. Es beginnt mit einem kurzen, offen fiktiven Szenario. Dann erklärt es das Problem und warum der naheliegende Ansatz zu kurz greift, stellt die Kontrolle vor und nennt ihre Kosten und die Fälle, in denen man sie nicht einsetzen sollte. Es listet häufige Fehler auf, die als Fallen im Design behandelt werden und nie als persönliches Versagen, und endet mit fünf bis sieben Erkenntnissen und einer Brücke zum nächsten Kapitel. Die gemeinsame Gestalt erlaubt Ihnen, die Kapitel

der Reihe nach zu lesen oder zu dem zu springen, das zum Problem dieser Woche passt, und den Abschnitt über die Kosten trotzdem dort zu finden, wo Sie ihn erwarten.

Lernen Sie Ledgerly kennen

Ledgerly ist ein fiktiver Service für Rechnungsstellung und Zahlungen zwischen Unternehmen. Firmen stellen darüber Rechnungen aus, ziehen Zahlungen ein und führen ein Hauptbuch darüber, was bewegt wurde. Er besteht aus fünf Services: `billing-api`, `invoice-worker`, `payments-service`, `reporting-job` und `admin-console`. Die Primary-Datenbank hat zwei Replikate und eine Analytics-Kopie für das Data-Team. Sie speichert Kontaktdaten von Kunden, Bankkontoinformationen und Zahlungs-Tokens, weshalb ihr Zugriffsdesign wichtig ist.

Die Menschen darum herum füllen vertraute Rollen aus. Es gibt Backend-Entwickler in Rufbereitschaft, ein Support-Team, ein Data-Team, eine kleine Plattformgruppe, die die Datenbanken betreibt, und einen externen Buchhaltungs-Vendor, der Exporte erhält.

Zu Beginn des Buches hat Ledgerly ein gemeinsames Admin-Passwort, Dauerzugriff für den Großteil des Backend-Teams, Desktop-Clients auf Laptops und keine Aufzeichnung darüber, wer was getan hat. Am Ende hat es eine Rolle für jeden Service, kurzlebige Credentials, ein Gateway für Menschen, einen geprüften Weg, Daten zu korrigieren, Anweisungen, die vor der Ausführung geprüft und begrenzt werden, maskierte Kopien und ein Log, das sich nicht unbemerkt umschreiben lässt. Es kommt dorthin Schritt für Schritt, und jedes Kapitel sagt, welchen Schritt es hinzufügt und was dieser Schritt das Team kostet.

Ledgerly ist erfunden, und alles, was dort geschieht, ebenso. Jedes Kapitel beginnt mit einem Szenario, das als Fiktion gekennzeichnet ist, und jede Zahl darin ist hypothetisch. Namen wie Robin, Sam, Jordan und Alex sind Platzhalter für Rollen in einem Team. Wenn in einem Szenario etwas schiefgeht, geht es in der Lehre um das Design, das es schiefgehen ließ, und nie um die Person in der Geschichte.

Wie Sie den Code lesen

Kein Beispiel in diesem Buch ist in einer echten Programmiersprache geschrieben. Prozeduren sind in schlichtem Pseudocode formuliert, und Rollen, Grants und Tabellen verwenden eine SQL-artige Notation, die zu keiner bestimmten Engine gehört. Nichts im Buch ist dafür gedacht, in ein Terminal eingefügt zu werden. Sie finden keine Konfigurationssyntax für einen bestimmten Proxy, Vault oder ein bestimmtes Cloud-Produkt, weil sich diese Syntax ändert und das Pattern dahinter nicht.

Diagramme verwenden Mermaid. Jedem geht ein Satz voraus, der Ihnen sagt, worauf Sie achten sollen, und verweigerte oder fehlschlagende Pfade sind beschriftet. Beispiele enthalten nie realistische Secrets, Kontonummern oder E-Mail-Adressen; Platzhalter wie `<secret>` und `IBAN_PLACEHOLDER` stehen an ihrer Stelle.

Warum Englisch und warum unabhängig von der Datenbank-Engine

Das Buch ist auf Englisch geschrieben, weil der größte Teil des Vokabulars dieses Feldes, von Least Privilege bis Break-Glass, auch in Teams, die in anderen Sprachen arbeiten, auf Englisch verwendet wird, und ich wollte, dass die Begriffe wiedererkennbar bleiben. Die Unabhängigkeit von der Engine hat einen verwandten Grund. Ein Team könnte PostgreSQL, MySQL oder etwas anderes betreiben und einen Cloud-Identitätsdienst, einen Secrets-Manager oder ein selbst gebautes Skript nutzen. Die Patterns sind in allen Fällen dieselben: den Eigentümer von den Nutzern trennen, jedem Workload eine eigene Identität geben, Credentials ablaufen lassen, Menschen einen identitätsbewussten Eingang vorschalten, prüfen, was läuft, bevor es läuft, Schreibvorgänge über etwas leiten, das sie ablehnen kann, aufzeichnen, was geschieht, und es überprüfen. Produkte und ihre Syntax ändern sich oft. Wenn ein echtes Produkt genannt wird, dann als Beispiel, und sein Verhalten wird so beschrieben, dass es über Versionen hinweg gelten sollte.

Ein Hinweis zur Gewissheit

Ich habe versucht zu sagen, welche meiner Empfehlungen fest sind und welche Vorlieben. Manche sind fest, weil ein Irrtum die Offenlegung von Kundendaten oder den Verlust des Hauptbuchs bedeuten würde, und ich sage es, wenn das der Fall ist. Die meisten sind Ermessensfragen, die ein nachdenkliches Team je nach Größe, Aufsichtsbehörden, Kunden und Häufigkeit der Alarmierungen anders gewichten könnte. Wo ich es nicht weiß, sage ich „Ich weiß es nicht“ und nenne die Variablen, die die Frage entscheiden würden.

Wenn Sie einer Empfehlung widersprechen, möchte ich, dass Sie die Prämisse finden können, der Sie widersprechen. Deshalb legt jedes Kapitel seine Begründung und seine Kosten offen. Ein Design, das Sie verstehen und an Ihre eigenen Randbedingungen angepasst haben, ist mehr wert als eines, das Sie aus einem Buch abgeschrieben haben.

Kapitel 1 beginnt mit der Tür selbst.

TEIL I

Grundlagen

Kapitel 1 — Die Tür, die niemand entworfen hat

Betrachten wir Ledgerly in seinen frühen Jahren. Ein potenzieller Kunde schickt einen Sicherheitsfragebogen herüber, und eine Zeile wirkt Routine: „Nennen Sie alle Personen und Systeme, die sich mit Ihrer Produktionsdatenbank verbinden können.“ Alex, der oder die neben anderen Aufgaben die Datenbanken betreut, nimmt sich der Sache an einem ruhigen Nachmittag an.

Alex beginnt beim Passwortmanager, der einen einzigen Admin-Eintrag enthält, den das ganze Backend-Team kennt. Als Nächstes kommen die fünf Services, die alle dasselbe Passwort gespeichert haben, und die Laptops, auf denen Desktop-Datenbank-Clients es sich merken. Dann die Analytics-Kopie, eine nächtliche Wiederherstellung eines unmaskierten Dumps, und einige ältere Dumps im gemeinsamen Speicher, von denen niemand weiß, wer sie angelegt hat. Jeder Ort, den Alex prüft, verweist auf einen weiteren.

Der Fragebogen hat eine zweite Zeile: „Woher wissen Sie, was diese getan haben?“ Alex öffnet das Datenbank-Log und findet auf jeder Zeile denselben Rollennamen, ganz gleich, ob die Anweisung von einem Service kam, von einem Entwickler, der eine Zeile korrigierte, oder von einem Report, der eigentlich ein Replikat hätte lesen sollen. Die ehrliche Antwort auf die erste Frage ist ein Haufen von Orten, und die ehrliche Antwort auf die zweite ist, dass Ledgerly es nicht weiß.

Alex braucht kein besseres Passwort. Alex muss wissen, woraus Zugriff besteht und wie eine kurze, vertretbare Antwort aussähe. Das ist die Frage dieses Kapitels: Woraus besteht der Zugriff auf eine Produktionsdatenbank, und was bräuchte Ledgerly, um diesen Fragebogen ohne eine Woche Nachforschung zu beantworten?

Dieses Kapitel beschreibt den Ausgangszustand von Ledgerly, wie es auch Kapitel 2 tut, und fügt noch keinen Schritt des Zielzustands hinzu. Seine Aufgabe ist es, Ihnen das Vokabular und die Karte für die folgenden Schritte zu geben.

Das Problem und warum es schwierig ist

Der Zugriff auf eine Produktionsdatenbank wird selten entworfen. Er sammelt sich an. Der erste Service braucht einen Weg hinein, also legt jemand eine Admin-Rolle und ein Passwort an. Der zweite Service verwendet sie wieder, weil es beim ersten funktioniert hat. Ein Entwickler muss sich eine Zeile ansehen, also kommt das Passwort in einen Desktop-Client. Ein geplanter Job braucht Daten und bekommt denselben Connection-String wie alles andere. Keiner dieser Schritte ist eine Entscheidung über Zugriff, und ihre Summe ist die Tür, die der Kapiteltitel meint.

Sehen Sie sich an, wie die Tür von Ledgerly aussieht, wenn Alex zu zählen beginnt. Das gemeinsame Admin-Passwort liegt in einem Passwortmanager des Teams, und nehmen wir zur Veranschaulichung an, dass etwa zwanzig Personen es kennen. Die Admin-Rolle, die es öffnet, ist Eigentümer jeder Tabelle. Alle fünf Services verbinden sich mit dieser Rolle, mit demselben Passwort, das in der Konfiguration jedes Services gespeichert ist, und auch die Deployment-Pipeline führt Migrationen damit aus. Entwickler erreichen den Primary von ihren Laptops aus über das Firmen-VPN, mit Clients, die sich das Passwort merken und jede Anweisung sofort nach der Ausführung mit einem Commit abschließen. Der Zugriff ist ein Dauerzugriff: Niemand verliert ihn je, auch Personen nicht, die vor einem Jahr in ein anderes Team gewechselt sind.

Die Replikate akzeptieren denselben Admin-Login, und Reporting-Abfragen laufen manchmal auf dem Primary. Die Analytics-Kopie ist eine Wiederherstellung eines unmaskierten Dumps, und die Dump-Dateien bleiben liegen. Ein externer Buchhaltungs-Vendor verbindet sich nie mit einer Datenbank von Ledgerly, erhält aber Exporte von Rechnungs- und Hauptbuchdaten, und der Export-Schritt von `reporting-job` schreibt diese Dateien mit dem gemeinsamen Admin-Login. Das Datenbank-Log zeigt eine einzige Rolle. Nichts an dieser Anordnung zeichnet auf, welche Person welche Anweisung ausgeführt hat.

Es ist verlockend, das als Frage der Disziplin zu behandeln: den Leuten sagen, sie sollen vorsichtig sein, das Passwort häufiger ändern. Ich würde dort nicht anfangen. Eine Regel, die nur funktioniert, wenn sich alle an sie erinnern, scheitert beim ersten Mal, wenn die Person in Rufbereitschaft müde ist. Das System gibt jedem Principal dieselben Fähigkeiten und denselben Namen, sodass jemand, der vorsichtig sein will, nichts hat, womit er vorsichtig sein könnte: keine engere Rolle zur Auswahl, keine Möglichkeit, der Datenbank „nur lesend“ zu sagen, keine Session, die einen Namen trägt.

Dasselbe gilt für das Mittel, nach dem man zuerst greift, nämlich das Passwort besser zu behandeln: in einem Vault speichern, rotieren, nur an Personen herausgeben, die danach fragen. Das ist Credential-Verteilung, und es lohnt sich, aber es beantwortet nur eine Frage: wer das Secret besitzt. Der Zugriff auf Produktionsdatenbanken ist kein Problem der Credential-Verteilung; es ist ein Problem der Autorisierung, der Ausführung und der Zurechenbarkeit. Die Autorisierung fragt, was jeder Principal tun darf und auf welchem Weg. Die Ausführung fragt, wie jede Anweisung die Daten erreicht, ob sie vorher von irgendetwas geprüft wird und wie weit sie reichen kann. Die Zurechenbarkeit fragt, ob der Nachweis jede Anweisung einer Person und einem Antrag zuordnet. Ein perfekt gespeichertes Passwort klärt, wer das Secret besitzt, und sonst nichts: Wer es besitzt, kann trotzdem eine Anweisung ausführen, die niemand geprüft hat, und das Log kann trotzdem einen gemeinsamen Namen zeigen.

Deshalb ist das eine Frage des Engineerings und keine der Richtlinien. Eine Richtlinie legt fest, was geschehen soll. Ein Design bestimmt, was geschehen kann und was eine Aufzeichnung hinterlässt, wenn es geschieht. Der Rest dieses Kapitels hat die Form eines Designproblems: woraus Zugriff besteht, was er leisten soll und wo jeder Teil in diesem Buch behandelt wird.

Zugriff besteht aus fünf Teilen

Bevor man ein Design beurteilt, hilft es, genau zu definieren, was entworfen wird. Ich finde es nützlich zu sagen, dass der Zugriff auf eine Produktionsdatenbank fünf Teile hat: eine Identität, einen Grant, einen Pfad, eine Ausführung und eine Aufzeichnung. Die Tabelle sagt, welche Frage jeder Teil beantwortet und zu welcher Säule der These des Buches er gehört.

Teil	Frage	Säule
Identität	Wer oder was handelt: eine Person oder eine Workload-Instanz?	Zurechenbarkeit
Grant	Was darf dieser Principal überhaupt tun: Rollen, Privilegien, Obergrenze?	Autorisierung
Pfad	Wie kommt die Verbindung zustande: Netzwerkreichweite, Gateway, Pooler?	Autorisierung (wo)
Ausführung	Wie erreicht eine Anweisung die Daten? Wer oder was reicht sie ein, wird sie vor der Ausführung geprüft, wie weit kann sie reichen (Transaktion, Row Guard, Timeout) und wie wird sie mit einem Commit abgeschlossen.	Ausführung
Aufzeichnung	Welcher Nachweis bleibt, und ordnet er die Anweisung der Person, dem Antrag und dem Ergebnis zu?	Zurechenbarkeit

Die drei Säulen bilden sich wie folgt auf die Teile ab. Die Autorisierung ist Grant plus Pfad, entschieden pro Anfrage. Die Ausführung ist das, was mit jeder Anweisung geschieht. Die Zurechenbarkeit ist Identität plus Aufzeichnung. Ein Passwort berührt nur die erste davon, und nur an der Tür.

Die **Identität** ist die verifizierte Behauptung, dass ein Principal der ist, der er zu sein vorgibt. Ein **Principal** ist alles, was sich authentifiziert: eine Person, ein Service, ein geplanter Job, eine Vendor-Integration. Principal ist das Wort, das man verwendet, wenn eine Aussage für Menschen und Workloads gleichermaßen gilt, und dieses Buch verwendet es durchgehend so. Eine Rolle ist keine Identität. Eine **Rolle** ist ein benanntes Bündel von Privilegien innerhalb der Datenbank, und viele Principals können sie sich teilen, genau wie es die gemeinsame Admin-Rolle tut.

Der **Grant** ist das, was der Principal tun darf, sobald er drin ist. Hier ist ein Grant der Vorgang, ein Privileg an eine Rolle oder eine Rolle an einen **Login** zu vergeben, und der daraus entstehende Eintrag; Kapitel 3 erweitert die Definition. Ein **Privileg** ist eine einzelne konkrete Sache, etwa das Lesen einer Tabelle, und der Login ist das Konto auf Datenbankebene, mit dem sich ein Principal verbindet. Der **Pfad** ist die Route, die eine Verbindung zur Datenbank nimmt: ein Netzwerk, ein Proxy, ein Gateway, ein Laptop.

Die **Ausführung** ist der Weg, auf dem eine Anweisung die Daten erreicht: wer oder was sie einreicht, ob vor der Ausführung irgendetwas sie prüft, was ihre Wirkung begrenzt und wie sie mit einem Commit abgeschlossen wird. Ein Grant sagt, dass eine Rolle `invoices` aktualisieren darf. Er sagt nicht, dass dieses Update eine einzige Zeile berühren soll, dass jemand es vorher angesehen hat oder dass es in einer Transaktion läuft, die sich zurückrollen lässt. Im Ausgangszustand beantwortet nichts diese Fragen: Der Desktop-Client reicht ein, was getippt wird, und schließt es auf der Stelle mit einem Commit ab. Kapitel 9 handelt von diesem Teil.

Die **Aufzeichnung** ist der Nachweis hinterher, die Antwort auf „Wer hat was wann getan?“ und, bei einem Schreibvorgang, „an welchen Zeilen, in welchem Antrag?“

Jeder Teil kann für sich versagen, und doch ist jeder auf die anderen angewiesen, um nützlich zu sein. Ohne eine eigene Identität hat ein präziser Grant niemanden, dem er zugeordnet werden kann. Ohne einen gewählten Pfad lässt sich eine Identität über eine Route umgehen, die nie nach einer fragt. Ohne kontrollierte Ausführung erlaubt ein enger Grant trotzdem eine Anweisung, die weiter reicht, als jemand wollte. Ohne Aufzeichnung lässt sich nicht erkennen, ob die anderen vier funktioniert haben. Diese Abhängigkeit ist der Grund, warum es den Rest des Problems bestehen lässt, wenn man nur einen Teil repariert, meist das Passwort.

Ein Test: Kann das Log die Frage beantworten?

Eine Möglichkeit herauszufinden, welche Teile fehlen, besteht darin, dem Log eine Frage zu stellen und zu beobachten, wo die Antwort aufhört. Der folgende Pseudocode verfolgt eine einzelne Anweisung durch die fünf Teile und nutzt dabei nur das, was der Ausgangszustand aufzeichnet.

```
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 }
```

Im Ausgangszustand von Ledgerly liefert `principalsUsing("admin")` etwa zwanzig Personen, fünf Services und die Deployment-Pipeline zurück, sodass die Funktion bei jeder Anweisung an der ersten Prüfung hält. Das ist kein Fehler im Log. Das Log ist wahrheitsgemäß über das, was es gesehen hat, nämlich einen einzigen Login. Die Lücke liegt im Design, das viele Principals hinter einen Login gestellt hat. Würde man die Identität beheben, hielte dieselbe Funktion als Nächstes an der Prüfung der Ausführung. Der Gründungsvorfall ist genau der Fall, für den sie da ist: Eine Korrektur einer einzelnen Zeile, deren Klausel `WHERE` einen ganzen Batch trifft, hat ein `rowsAffected` weit über `expectedRows`, wurde nie geprüft und zeigte ihre Zahl nie vor dem `COMMIT`, sodass sie bei „execution“ hält, obwohl die Person bekannt ist und der Grant es erlaubte. Eine Anweisung, die diese Prüfung besteht, kann trotzdem bei „record“ hängen bleiben, wenn nichts sie einem Antrag und einem Ergebnis zuordnet.

Anwendungen und Menschen, eine Frage

Wenn Menschen „Datenbankzugriff“ hören, denken sie oft an Entwickler mit Abfrage-Tools. Dieses Bild ist auf eine Weise unvollständig, die zählt. Ein Großteil des Verkehrs zum Primary von Ledgerly kommt von Software: `billing-api` bedient die Web-App, `invoice-worker` verschickt Erinnerungen, `payments-service` spricht mit Zahlungsanbietern, `reporting-job` läuft nachts, und `admin-console` handelt für Mitarbeitende im Support. Software verbindet sich öfter, länger und mit weniger Aufsicht als jeder Mensch.

Services und Menschen fallen auf unterschiedliche Weise aus, deshalb behandelt Teil II die Anwendungen und Teil III die Menschen. Der Kontrast ist konkret:

	Services	Menschen
Häufigkeit	Ständig, rund um die Uhr	Gelegentlich, in Schüben
Aufsicht	Keine, solange sie laufen	Eine Person beobachtet das Ergebnis
Lebensdauer der Credentials	Kurzlebig und von einem Broker ausgestellt (Kapitel 4)	Eine Session oder ein genehmigtes Zeitfenster (Kapitel 7 und 8)
Typischer Ausfall	Ein offengelegtes Secret, ein erschöpfter Connection-Pool, ein Deployment, das auf die falsche Datenbank zeigt	Eine Anweisung, die weiter reicht als gedacht, unter Druck von Hand ausgeführt

Beide beruhen auf den Grundlagen in Teil I.

Es gibt noch eine Art von Principal, die leicht übersehen wird: einen Service, der im Auftrag einer Person handelt. `admin-console` wird vom Support genutzt, verbindet sich im Ausgangszustand aber wie alles andere mit dem gemeinsamen Admin-Login, sodass die Datenbank die Konsole sieht und nie die Person, die geklickt hat. Sam im Support sieht sich jeweils einen Kunden an, und das Design lässt die Konsole die Daten aller Kunden unter einem Namen erreichen, der niemandes ist. Kapitel 3 zeigt, wie die Konsole die Identität der Person mit jeder Aktion weitergibt, und Kapitel 14, wie die Audit-Einträge zusammengeführt werden, sodass die Person den Hop überlebt.

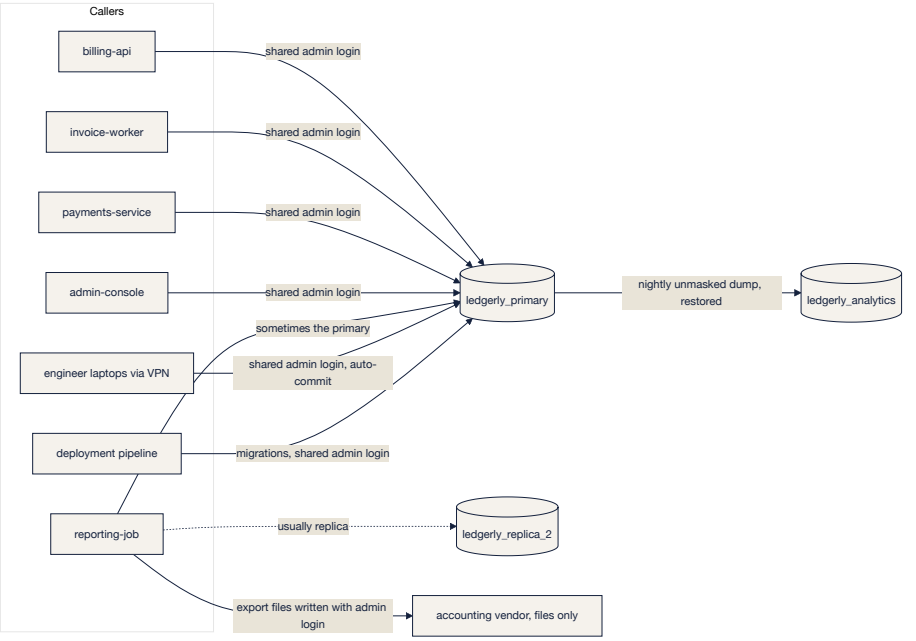
Sicherheit und Zuverlässigkeit sind dieselbe Frage

Teams führen oft zwei Listen. Die eine handelt von Sicherheit: wer Daten stehlen, wer Zugriff missbrauchen könnte. Die andere handelt vom Betrieb: was die Datenbank lahmlegen oder beschädigen könnte. Sie gehören meist verschiedenen Personen und werden zu verschiedenen Zeiten geprüft. Für den Zugriff würde ich sie zusammenlegen, denn die Fragen sind dieselben.

Nehmen wir drei Ereignisse, die zusammenhanglos klingen. Eine Anweisung mit einer falschen oder fehlenden Klausel `WHERE` aktualisiert alle Rechnungen statt einer. Ein Laptop-Client lässt eine Verbindung offen und belegt den ganzen Abend einen Slot. Ein Dump der Produktionsdatenbank wird in den gemeinsamen Speicher kopiert und von jemandem gelesen, der dort nichts zu lesen hatte. Das erste ist ein Problem der Datenintegrität, das zweite eines der Verfügbarkeit, das dritte eines der Vertraulichkeit. Und doch fragt jedes dasselbe vom Design: Warum konnte ein einzelner Principal etwas so Weitreichendes tun, ohne eine engere Route, ein Limit oder eine Aufzeichnung?

Die drei Ereignisse haben eine gemeinsame Ursache im Design: keine Verengung von Grant, Lebensdauer oder Pfad, und nichts, was hinterher etwas zeigt. Die falsche Klausel `WHERE` ist überwiegend eine Lücke bei Grant und Ausführung, die offene Verbindung eine Lücke bei Lebensdauer und Limit, und der kopierte Dump eine Lücke beim Pfad. Den Grant verengen, die Lebensdauer begrenzen, den Pfad wählen, die Anweisung begrenzen und die Aufzeichnung führen: Das hilft bei allen drei Ereignissen zugleich. Darum bedient ein einziges Design beide Listen. Eine Rolle, die `payment_tokens` nicht lesen kann, schützt Kunden vor einem Abfluss, und eine Rolle, die `ledger_entries` nicht schreiben kann, schützt das Hauptbuch vor einem fehlerhaften Batch-Update. Beides entspringt derselben Gewohnheit, eng zu vergeben.

Das Diagramm unten zeigt den Ausgangszustand von Ledgerly. Achten Sie darauf, wie viele Pfeile am selben Knoten enden und welcher eine Pfeil ihn zur Analytics-Kopie hin verlässt.



Zwei Dinge fallen auf. Erstens trägt jeder Pfad zum Primary denselben Namen, sodass die Datenbank einen Worker nicht von einem Menschen unterscheiden kann. Zweitens verlassen die Daten das System auch über eine Seitenroute: Die nächtliche Wiederherstellung kopiert alles, auch Spalten, die kein Analyst braucht, in einen Speicher mit eigenem Publikum.

Die Bandbreite der Zugangswege

Zwischen dem gemeinsamen Passwort und einem vollständig kontrollierten Design liegt eine Reihe von Patterns, und die späteren Kapitel ordnen jedes darauf ein. Das Diagramm verläuft von der geringsten zur größten Kontrolle über jede Anweisung. Achten Sie darauf, was jeder Schritt hinzufügt und wo die erste Prüfung auf Ebene der Anweisung erscheint.



Jeder Schritt nach rechts fügt eine Eigenschaft hinzu und kostet etwas. Eine Bastion verengt den Pfad, aber die Session auf der anderen Seite bleibt unbeaufsichtigt. Ein Session-Proxy bindet eine Verbindung an eine Person und zeichnet sie auf, übergibt aber eine Session, in der alles getippt werden kann, was der Grant erlaubt. Ein Gateway, das die Ausführung

vermittelt, sieht sich jede Anweisung vor der Ausführung an und kann sie ablehnen. Ein Produktpfad beseitigt die Session ganz: Die Person bittet um eine definierte Operation, etwa „diese Rechnung erneut senden“, und schreibt nie SQL. Nichts davon besagt, dass das Pattern ganz rechts immer das beste ist; es ist auch das aufwendigste im Bau und das langsamste in der Änderung. Kapitel 7 vergleicht die ersten vier auf konkreten Pfaden, Kapitel 9 behandelt die Prüfungen auf Ebene der Anweisung, und das Schlusskapitel bewertet die Patterns nach Säulen.

Wie guter Zugriff aussieht

Ich würde das Ziel aufschreiben, bevor ich irgendein Tool wähle. Ich verwende fünf Eigenschaften, und jede hat einen Preis.

Minimale Rechte sind die Praxis, nur zu vergeben, was eine Aufgabe braucht, und nur so lange, wie sie es braucht. Der Preis ist der Aufwand, herauszufinden, was jede Aufgabe braucht, und die Reibung, wenn sich zeigt, dass eine Aufgabe mehr braucht. Ein Team, das sie einführt, ohne einen schnellen Weg, um mehr zu bitten, wird zusehen, wie Leute sie umgehen.

Kurze Lebensdauer bedeutet, dass Zugriff von selbst endet. Ein Grant, der eine Stunde hält, braucht niemanden, der sich an ihn erinnert. Dem steht der **Dauerzugriff** gegenüber, der ununterbrochen besteht, ob er genutzt wird oder nicht. Ein Dauer-Grant ist ein Versprechen, dass niemand je wieder fragen muss, und im ersten Monat ist er bequem und in jedem Monat danach teuer, weil sich niemand erinnert, was das Versprechen abdeckte. Der Preis der kurzen Lebensdauer ist häufigeres Fragen und ein Ablauf, der mitten in einer Reparatur eintreffen kann.

Observability bedeutet, dass jede Nutzung von Zugriff eine Aufzeichnung hinterlässt, die den Principal nennt, und dass die Aufzeichnung an einem Ort liegt, den der Principal nicht ändern kann. Der Preis sind Speicher und, wichtiger, Aufmerksamkeit: Ein Log, das niemand liest, ist Sicherheitstheater, denn es kostet Speicher und schützt nicht. Kapitel 14 kommt darauf zurück.

Umkehrbarkeit hat zwei Bedeutungen, und beide zählen. Zugriff sollte sich leicht entziehen lassen, und ein Fehler, der mit Zugriff gemacht wurde, sollte sich rückgängig machen oder zumindest eingrenzen lassen. Der

Entzug muss Routine sein, und das Rückgängigmachen eines Fehlers braucht geprüfte Korrekturen und Backups, die sich wiederherstellen lassen. Eine günstige Form der Umkehrbarkeit ist, einen Schreibvorgang in einer expliziten Transaktion auszuführen und die Zeilenzahl vor dem `COMMIT` zu zeigen: Ein Update, das eine einzige Zeile berühren sollte und, sagen wir, vierhundert meldet, wird gestoppt, solange es sich noch zurückrollen lässt. Kapitel 9 behandelt, wie ein Gateway das zur Voreinstellung macht. Der Preis ist die Zeit, die für Übungen aufgewendet wird.

Geringe Reibung ist die Eigenschaft, die Teams zu notieren vergessen. Wenn der legitime Weg langsam ist, finden die Leute den illegitimen, und sie haben recht damit, denn der Incident um 2 Uhr nachts wartet nicht. Der Preis der geringen Reibung ist, dass sie die anderen vier einschränkt: Jede Kontrolle muss billig genug sein, um in der schlimmsten Stunde der Woche benutzt zu werden.

Diese fünf ziehen in verschiedene Richtungen. Kurze Lebensdauer erhöht die Reibung. Observability fügt einen Hop hinzu: Eine aufgezeichnete Session muss durch etwas laufen, das sie aufzeichnet, und dieses Etwas kann langsam sein oder ausfallen. Minimale Rechte vervielfachen die Rollen. Ich glaube nicht, dass es eine Einstellung gibt, die alle fünf maximiert. Die nützliche Frage lautet nicht „Wer sollte Zugriff haben?“, sondern „Welche Anweisungen dürfen laufen, worüber und mit welcher Aufzeichnung?“ Jedes spätere Kapitel beantwortet einen Ausschnitt davon und sagt, was die Antwort kostet.

Eine erste Bestandsaufnahme

Bevor Sie irgendeine Kontrolle einführen, würde ich aufschreiben, was existiert. Die folgende Skizze zeigt die Form einer Bestandsaufnahme für den Ausgangszustand von Ledgerly. Es geht darum, eine Zeile pro Tür zu haben, mit einem benannten Verantwortlichen und einem Grund, sodass eine Zeile ohne Verantwortlichen als Lücke sichtbar wird.

```
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
```

Füllen Sie einen Block pro Tür aus: jeden Service, den geplanten Job, die Deployment-Pipeline, die Laptops, die Analytics-Wiederherstellung, die Dumps im gemeinsamen Speicher und die Export-Dateien. Die meisten Teams stellen fest, dass die Zeilen `owner` und `ends` die sind, die leer bleiben. Eine solche Bestandsaufnahme ist kein Projektplan. Sie ist das, was „Wer kann den Primary erreichen?“ von einem Tag Nachforschung in ein Dokument verwandelt, das man lesen kann.

Eine Karte dieses Buches

Die folgenden Kapitel nehmen jeden Teil des Zugriffs und jeden der obigen Ausfälle der Reihe nach vor. Die Tabelle verknüpft jede Frage, die dieses Kapitel aufgeworfen hat, mit der Stelle, an der das Buch sie beantwortet.

Die Frage	Säule	Wo sie beantwortet wird
Wovor und vor welchen Arten von Ereignissen schützen wir Daten?	Alle drei	Kapitel 2
Wie geben wir jedem Principal eine eigene Identität und die engste Rolle, die funktioniert?	Autorisierung	Kapitel 3
Wie bekommen Services Credentials ohne gemeinsames Passwort in einer Datei?	Zurechenbarkeit (Identität)	Kapitel 4
Wie werden Verbindungslimits und Timeouts zu einer Richtlinie, die die Datenbank schützt?	Autorisierung und Ausführung (Begrenzungen)	Kapitel 5
Wie erreichen Lesezugriffe Replikate, und was geschieht mit den anderen Kopien der Daten?	Autorisierung	Kapitel 6

Die Frage	Säule	Wo sie beantwortet wird
Auf welchen Pfaden erreichen Menschen die Datenbank, und was liegt auf jedem Pfad?	Autorisierung (Pfad), Zurechenbarkeit	Kapitel 7
Wie bekommt jemand Schreibzugriff für eine Aufgabe, und wie endet er?	Autorisierung	Kapitel 8
Wie wird eine Anweisung vor der Ausführung geprüft, und wie wird sie mit einem Commit abgeschlossen?	Ausführung	Kapitel 9
Wie bleiben Ad-hoc-Abfragen und Datenkorrekturen überprüfbar?	Ausführung	Kapitel 10
Wer ändert das Schema, und mit welchem Zugriff?	Ausführung	Kapitel 11
Welche Daten brauchen zusätzlichen Schutz, und wie werden sie maskiert oder zurückgehalten?	Autorisierung	Kapitel 12
Was ist mit Backups, Exporten und den anderen Routen an den Kontrollen vorbei?	Autorisierung (Pfad)	Kapitel 13
Wie führen wir eine Aufzeichnung, die standhält, und überprüfen sie?	Zurechenbarkeit	Kapitel 14
Was ändert sich bei einem Incident?	Alle drei, unter Druck	Kapitel 15
Was tun wir zuerst, und wie halten wir es durch?	Alle drei	Kapitel 16

Das Buch hat fünf Teile: Grundlagen, Zugriff durch Anwendungen, Zugriff durch Menschen, Schutz der Daten und ihrer Kopien sowie Nachweis und Betrieb von alledem. Jedes Kapitel nennt zu Beginn die Säule, der es dient. Sie müssen nicht der Reihe nach lesen. Wenn Ihr unmittelbares Problem das gemeinsame Passwort in fünf Konfigurationsdateien ist, ist Kapitel 4 der Ort zum Anfangen, aber Kapitel 3 bewahrt Sie davor, aus Versehen die nächste gemeinsame Rolle anzulegen.

Abwägungen und Kosten

Die erste Kostenart sind **Zeit und Aufmerksamkeit einer kleinen Gruppe**. Jemand muss die Bestandsaufnahme erstellen, Rollen definieren, Credentials einrichten und Reviews durchführen. In einem kleinen Team

ist das oft die Teilzeitaufgabe einer einzigen Person, und sie konkurriert mit der Produktarbeit. Wird die Arbeit nie eingeplant, bleibt das Design auf dem Papier. Ich sähe lieber einen kleinen, abgeschlossenen ersten Schritt als einen großen Plan.

Die zweite Kostenart ist die **Reibung für die Person in Rufbereitschaft**. Jede Kontrolle stellt einen Schritt zwischen eine müde Person und eine Reparatur. Enge Rollen führen dazu, dass der erste Versuch im schlechtesten Moment manchmal mit einem Privilegienfehler scheitert. Kurzlebiger Zugriff bedeutet, erneut zu fragen. Legen Sie vorab fest, was der schnelle Weg ist, und üben Sie ihn. Eine Kontrolle, die bei jedem Incident umgangen wird, ist keine Kontrolle; sie ist eine Verzögerung vor der Umgehung.

Die dritte Kostenart sind **neue Systeme, die verfügbar sein müssen, wenn man sie braucht**. Ein Gateway, ein Speicher für Secrets und ein Log-Store sind jeweils etwas weiteres, das ausfallen kann. Fragen Sie sich vor der Einführung, was mit der Person geschieht, die während dieses Ausfalls die Datenbank braucht, und halten Sie eine Antwort bereit.

Es gibt auch Fälle, in denen das volle Design der falsche Ausgangspunkt ist. Ein Prototyp mit einem einzigen Service und ohne Kundendaten, betrieben von zwei Personen, braucht weder ein Gateway noch einen Review-Kalender. Er braucht aber einen eigenen Login, getrennt von dem jeder Person, weil das billig ist, und eine Notiz, die festhält, was das Team dazu bringen würde, die Frage neu zu stellen: der erste Kunde, der dritte Entwickler oder die erste externe Integration. Ich kenne den richtigen Zeitpunkt für Ihr Unternehmen nicht. Er hängt davon ab, und zwar wovon: wie sensibel die Daten sind, wie viele Principals sie berühren und wie schnell eine fehlerhafte Anweisung einen Schaden anrichten kann, den Sie nicht rückgängig machen können.

Schließlich haben die Eigenschaften guten Zugriffs eine Grenze. Sie verkleinern den Auswirkungsbereich und machen Verhalten sichtbar. Sie machen eine Datenbank nicht sicher, und kein Kapitel dieses Buches behauptet etwas anderes. Der **Auswirkungsbereich** ist das Ausmaß dessen, was betroffen sein kann, wenn ein Principal, ein Credential oder eine Aktion schiefgeht, und die Arbeit dieses Buches besteht darin, diesen Bereich kleiner und sichtbarer zu machen, nicht ihn auf null zu bringen.

Häufige Fehler

Das sind Fallen im Design, kein persönliches Versagen. Jede ist aus einem Grund verlockend, und jede hat ein Gegenmittel, das nicht davon abhängt, dass jemand vorsichtiger ist.

- **Eine Passwortänderung als Gegenmittel behandeln.** Ein neues Passwort fühlt sich nach Fortschritt an, und es entzieht dem früheren Kollegen sein Wissen. Aber wenn sich, sagen wir, zwanzig Personen und fünf Services es weiterhin teilen, ist das Design unverändert. Die Lösung ist ein Design, in dem niemand ein Passwort teilen muss, und genau das bauen die Kapitel 3 und 4 auf.
- **Den Grant reparieren und den Pfad vergessen.** Enge Rollen helfen nur, wenn Verbindungen sie benutzen müssen. Wenn Laptops den Primary weiterhin mit einem alten Admin-Login erreichen können, sind die engen Rollen optional. Erfassen Sie jeden Pfad, nicht nur jede Rolle.
- **Die Anwendungen auslassen.** Reviews der Frage „Wer hat Datenbankzugriff“ listen oft nur Personen auf, weil Personen im Organigramm stehen. Die Services halten die mächtigeren Credentials und nutzen sie öfter. Beziehen Sie jeden Workload, jeden geplanten Job und jedes Skript ein.
- **Den sicheren Pfad langsamer machen als den unsicheren.** Wenn das Lesen der Produktion ein Ticket und einen Tag braucht, während das gemeinsame Passwort, sagen wir, zehn Sekunden braucht, gewinnt das gemeinsame Passwort jedes Mal. Streben Sie einen sicheren Pfad an, der nicht langsamer ist.

> **Warnung:** Ein unmaskierter Dump im gemeinsamen Speicher ist der direkteste Weg des Ausgangszustands zu Kundendaten, und er steht jedem offen, der den Ordner lesen kann. Betrachten Sie das Finden und Entfernen dieser Dateien als Teil der ersten Bestandsaufnahme, noch vor jedem Umbau der Rollen.

Zentrale Erkenntnisse

- Der Zugriff auf Produktionsdatenbanken ist kein Problem der Credential-Verteilung; es ist ein Problem der Autorisierung, der Ausführung

und der Zurechenbarkeit, und ein Passwort beantwortet nur die erste Frage, an der Tür. Zugriff ist ein Design aus fünf Teilen, nämlich Identität, Grant, Pfad, Ausführung und Aufzeichnung, und eine Schwäche in einem von ihnen untergräbt die anderen.

- Ein gemeinsames Admin-Konto ist keine Entscheidung, die jemand getroffen hat; es sammelt sich an, und das Gegenmittel ist ein Design, in dem niemand ein Passwort teilen muss.
- Anwendungen verbinden sich öfter und mit weniger Aufsicht als Menschen, daher muss jede Behandlung von Zugriff Services, Jobs und Tools neben Menschen als Principals abdecken.
- Sicherheit und betriebliche Zuverlässigkeit sind für den Zugriff eine einzige Frage, weil eine falsche Klausel `WHERE`, eine offen gelassene Verbindung und ein kopierter Dump eine gemeinsame Ursache im Design haben: keine Verengung von Grant, Lebensdauer oder Pfad und keine Aufzeichnung.
- Guter Zugriff hat fünf Eigenschaften, nämlich minimale Rechte, kurze Lebensdauer, Observability, Umkehrbarkeit und geringe Reibung, und sie ziehen in verschiedene Richtungen, sodass jede Kontrolle mit ihren Kosten vorgestellt werden sollte.
- Der erste Schritt ist eine Bestandsaufnahme mit einer Zeile pro Tür, einschließlich der Services, der Laptops und der Datenkopien, bei der ein fehlender Verantwortlicher oder ein fehlendes Enddatum als sichtbare Lücke auffällt.
- Kontrollen verkleinern den Auswirkungsbereich und machen Verhalten sichtbar, aber sie machen eine Datenbank nicht sicher, und das ehrliche Ziel ist ein kleinerer Bereich und ein klarerer Blick.

Das nächste Kapitel wendet sich von der Gestalt des Zugriffs den Gründen zu, ihn zu schützen. Kapitel 2 fragt, wem und was die Kontrollen dienen, sortiert die Ereignisse, die schiefgehen können, in drei Arten und schreibt ein Bedrohungsmodell für Ledgerly, auf das sich der Rest des Buches zurückbeziehen kann.

Kapitel 2 — Ein Bedrohungsmodell für Produktionsdaten

Stellen Sie sich vor, Ledgerlys Sicherheitskontakt schickt dem Platform-Team vor einem Quartals-Review eine kurze Frage: „Vor wem schützen wir die Produktionsdatenbank?“ Alex, zuständig für den Betrieb der Datenbanken, antwortet binnen einer Minute: vor Außenstehenden, vor allem. Dann hält Alex inne, denn die Antwort passt nicht zu den eigenen Tickets des Teams aus dem vergangenen Jahr. Eine Support-Anfrage, die den falschen Kunden betraf. Eine Reporting-Abfrage, die die Rechnungsschreibvorgänge einen Nachmittag lang verlangsamte. Ein Dump, der für einen Tag Debugging auf einem Laptop wiederhergestellt wurde und den niemand daran erinnert hat zu löschen. Ein Export an einen Dienstleister, der mit einer Spalte mehr hinausging als beabsichtigt.

An keinem davon war ein Außenstehender beteiligt. Auch böse Absicht war nirgends im Spiel. Bei allen ging es um eine Tür, die jemandem offenstand, der eine vernünftige Aufgabe hatte, und um eine Rolle, die mehr erlaubte, als die Aufgabe brauchte. Die erste Antwort von Alex war die, die nach Sicherheit klingt. Die zweite Liste ist die, die das System beschreibt.

Dieses Kapitel beschreibt Ledgerly in seinem Ausgangszustand, wie schon Kapitel 1, und fügt keine eigene Kontrolle hinzu. Es fügt ein schriftliches Bedrohungsmodell hinzu, die Referenz, an der jede spätere Kontrolle des Buches ausgewählt wird. Es ordnet die Bedrohungen allen drei Säulen zu: was ein Principal tun darf, wie jede Anweisung die Daten erreicht und welcher Nachweis bleibt. Die Frage, die es beantwortet, ist die, die Alex nicht in einer Minute beantworten konnte: Wer und was kann den Daten schaden, durch welche Türen, und wie weit reicht der Schaden?

Das Problem und warum es schwierig ist

Ein Zugriffsdesign ohne Bedrohungsmodell folgt meist der lautesten Sorge. Ist das ein Angreifer von außen, baut das Team einen Perimeter: ein VPN, eine Firewall-Regel, eine Netzwerk-Allowlist. Auf diese kommt es an. Aber ein Perimeter beantwortet die Frage „Wer darf in die Nähe der Datenbank?“ und sagt nichts darüber, was eine Person oder ein Service tun kann, sobald sie drin ist. In Ledgerlys Ausgangszustand kann jeder im VPN mit dem gemeinsamen Passwort alles tun, was die Admin-Rolle kann, und das ist alles.

Die naheliegende Alternative ist eine Liste von Kontrollen aus einer Checkliste, der Reihe nach angewandt. Eine Checkliste weiß nicht, welche Ihrer Türen das größte Risiko tragen. Sie verlangt die Rotation eines Passworts, das niemand ohne Ausfall rotieren könnte, und sagt nichts über die nächtliche Wiederherstellung, die jede Restricted-Spalte in einen Speicher mit deutlich größerem Publikum kopiert. Kontrollen, die ohne Modell gewählt werden, werden nach Verfügbarkeit und Mode gewählt. Manche senken die Gefährdung. Manche sind Sicherheitstheater, Reibung ohne entsprechende Risikominderung, und das Team kann nicht unterscheiden, welche welche ist, weil es nie aufgeschrieben hat, wofür die Kontrolle gedacht war.

Ein schriftliches Modell wird aus drei Gründen gebraucht. Erstens sind die Türen technische Objekte: ein Login, ein Netzwerkpfad, eine Datei in einem gemeinsamen Speicher, eine Pipeline-Variable. Eine Richtlinie kann sagen „Zugriff muss angemessen sein“, aber nur ein Design entscheidet, was ein Login erreichen kann. Zweitens entstehen die größten Risiken aus dem Zusammenspiel gewöhnlicher Entscheidungen, wie das Vorwort beschreibt: ein gemeinsames Passwort, ein Client mit Auto-Commit und eine Rolle, der jede Tabelle gehört. Keine einzelne Entscheidung sieht falsch aus, also findet ein Review einzelner Entscheidungen das Problem nicht. Drittens muss das Modell günstig genug zu pflegen sein, sonst wird es einmal geschrieben und dem Drift überlassen.

Ein Bedrohungsmodell im hier verwendeten Sinn ist klein. Es listet auf, wer und was auf die Daten einwirken kann, über welchen Pfad, was schiefgehen könnte, wie weit sich die Wirkung ausbreitet und welche Kon-

trolle sie abdeckt. Es ist weder ein Risikowert noch eine Prognose. Es ist eine Möglichkeit, bei jeder Tür dieselbe Frage zu stellen, damit sich die Antworten vergleichen lassen.

> **Hinweis:** Das Buch beschreibt Bedrohungen nur so weit, wie sie eine Kontrolle rechtfertigen. Dieses Kapitel benennt, was schiefgehen kann, etwa ein Credential, das abfließt, oder eine Abfrage, die auf der falschen Datenbank landet, und beschreibt nicht, wie jemand es herbeiführen würde. Wenn mehr Detail das Design nicht ändern würde, bleibt es weg.

Wer und was die Datenbank erreicht

Kapitel 1 sagte, dass der Zugriff auf eine Produktionsdatenbank aus fünf Teilen besteht: einer Identität, einem Grant, einem Pfad, einer Ausführung und einer Aufzeichnung. Ein Bedrohungsmodell beginnt damit, die Principals aufzulisten und jedem die fünf Fragen zu stellen. Ledgerly hat neun Arten von Akteuren, die es wert sind, aufgelistet zu werden. Zwei davon, der Dienstleister und der Angreifer von außen, sind im strengen Sinn keine Principals, und genau deshalb gehören sie auf die Liste: Man lässt sie leicht weg.

Services

Die Services (`billing-api`, `invoice-worker`, `payments-service` und `admin-console`, wobei `reporting-job` als Nächstes als Job behandelt wird) sind die stärksten Nutzer der Datenbank. Sie laufen unbeaufsichtigt, halten ihre Credentials in der Konfiguration und verhalten sich nur so korrekt wie ihr Code. Ein Service kann sich fehlverhalten, ohne dass jemand etwas falsch macht: eine Retry-Schleife, die Verbindungen öffnet, bis das Limit erreicht ist, eine Abfrage, die eine Tabelle durchsucht, weil ein Index gelöscht wurde, ein Deployment, das einen Staging-Build auf den Primary zeigen lässt.

Im Ausgangszustand verbindet sich jeder Service mit demselben Admin-Login, sodass der Auswirkungsbereich jedes Services die ganze Datenbank ist, obwohl `billing-api` und `invoice-worker` jeweils nur eine Handvoll

Tabellen brauchen. Ein Defekt im kleinsten Service hat die Reichweite des größten.

Batch-Jobs

`reporting-job` verdient eine eigene Zeile, weil ein Job nach Zeitplan läuft, oft nachts, ohne dass jemand zusieht. Er liest tendenziell viele Daten, hält lange Transaktionen und erzeugt Dateien. Im Ausgangszustand liest er manchmal den Primary, und sein Export-Schritt schreibt die Dateien für den Dienstleister mit dem gemeinsamen Admin-Login. Sein normales Verhalten sieht schon wie Massenzugriff aus, was sein abnormales Verhalten schwerer erkennbar macht als das einer Person.

Entwickler in Rufbereitschaft

Die Rufbereitschaftsrotation steht am stärksten unter dem Druck, breiten Zugriff zu haben, und das aus gutem Grund: Eine Alarmierung mitten in der Nacht ist nicht der Moment, auf eine Genehmigung zu warten, die niemand wach ist zu erteilen. Im Ausgangszustand verwenden sie das gemeinsame Admin-Passwort über Desktop-Clients, die es sich merken und bei jeder Anweisung einen Commit ausführen. Der Zeitdruck beim Reparieren zusammen mit fehlender Reibung beim Ändern macht dies zur Gruppe, in der Unfälle am wahrscheinlichsten sind, und zur Gruppe, die das Design am besten schützen kann.

Support

Mitarbeitende im Support nutzen `admin-console`, um jeweils einen Kunden anzusehen. Die Bedrohung liegt hier im Umfang, nicht in böser Absicht. Ein Tool, das jeden Kunden anzeigen kann, gestützt auf eine Rolle, die jede Spalte lesen kann, beantwortet eine Frage zu einer einzigen Rechnung mit einem System, das jede Rechnung zeigen könnte. Die Designfrage lautet, wie sich die breite Fähigkeit unzugänglich machen lässt, wenn nur die schmale gebraucht wird.

Das Data-Team

Das Data-Team arbeitet auf `ledgerly_analytics`, im Ausgangszustand eine nächtliche Wiederherstellung eines unmaskierten Dumps: jede Restricted- und Confidential-Spalte, erreichbar für eine Gruppe, deren Aufga-

be es ist, Daten in großen Mengen anzusehen. Ihr Zugriff ist legitim. Das Problem ist, dass die Kopie ein größeres Publikum, lockerere Handhabung und weniger Logs hat als der Primary, sodass eine Kontrolle auf dem Primary nicht schützt, was in der Kopie liegt.

Dienstleister und Integrationen

Der externe Buchhaltungsdienstleister verbindet sich nie mit einer Ledgerly-Datenbank. Er erhält Dateien. Das verändert die Gestalt der Bedrohung: Die Gefährdung ist kein Login, sondern ein Export, und sein Auswirkungsbereich ist, welche Spalten der Export enthält und wie lange die Kopie auf Seiten des Dienstleisters existiert. Die relevanten Fragen sind, was die Datei enthält, wer sie danach verantwortet und ob jemand bemerken würde, wenn die Datei eine Spalte dazubekäme.

CI/CD

Die Deployment-Pipeline ist ebenfalls ein Principal, obwohl sie Infrastruktur ist und weder eine Person noch ein Service. Im Ausgangszustand führt sie Migrationen mit demselben Admin-Login aus, weil die Admin-Rolle jede Tabelle besitzt und nichts anderes sie ändern kann, und das Credential liegt in der Konfiguration der Pipeline. Eine Pipeline, die jede Tabelle ändern kann und ein dauerhaftes Credential hält, ist zugleich ein Kompromittierungsrisiko und ein Fehlerrisiko, denn eine fehlerhafte Migration läuft mit den vollen Privilegien des Eigentümers.

Platform- und DBA-Mitarbeitende

Die kleine Platform-Gruppe betreibt die Datenbanken, und sie ist der eine Akteur, der die Kontrollen selbst ändern kann: Rollen, Grants, den Eingang, durch den Menschen gehen, das Log. Im Ausgangszustand verwendet sie wie alle den gemeinsamen Admin-Login, sodass nichts ihre Arbeit von der aller anderen trennt. Im Zieldesign hält sie eine Rolle zur Verwaltung des Zugriffs, `ledgerly_access_admin`, die drei Principals über ihre eigenen Logins nutzen. Das beseitigt die Bedrohung nicht; es verschiebt sie. Ein fehlerhafter Grant oder eine abgeschaltete Log-Einstellung durch jemanden mit dieser Rolle ändert, was jede andere Zeile des Modells bedeutet.

Aus diesem Grund führt das Modell die **Control Plane** als eigenes Asset: das Access-Gateway, den Credential-Broker, `ledgerly_access_admin` und den Audit-Speicher. Die Daten werden von den Kontrollen geschützt, und die Kontrollen werden von einem viel kleineren Kreis an Personen und Systemen geschützt. Die Gefährdung ist im Umfang klein und in der Wirkung groß, und die Fragen sind dieselben wie bei jeder Tür: Wer kann sie erreichen, was könnte sie ändern, und wer würde es bemerken.

Der Angreifer von außen

Ein Angreifer von außen ist kein Principal, bis er ein Credential erlangt, also modelliert man ihn sinnvoll danach, was er erlangen müsste, nicht danach, was er tun könnte. Die Fragen der Verteidigung lauten: Welche Türen akzeptieren ein Credential, das abfließen könnte, was würde es erreichen, und wie würden wir es erfahren? Die Antworten kommen von den anderen acht Akteuren. Ein abgeflossenes Credential erbt den Auswirkungsbereich dessen, dem es gehörte.

Drei Bedrohungsklassen

Dieselben Akteure können auf drei verschiedene Arten Schaden anrichten, und die Kontrollen unterscheiden sich je Klasse. Wer die Klassen vermischt, kommt zu unpassenden Kontrollen: Eine Genehmigung der Absicht, entworfen für Missbrauch, hilft wenig gegen einen Unfall, weil der Genehmiger zustimmt, dass die Aufgabe legitim ist, und die Anweisung trotzdem schiefgeht.

Fehler

Ein Fehler ist eine Handlung, die korrekt sein sollte und es nicht war: eine Anweisung gegen die falsche Umgebung, eine Bedingung, die mehr Zeilen trifft als beabsichtigt, eine Abfrage, die für den Primary zu schwer ist, ein Skript, das eine veraltete Konfiguration auf die Produktion zeigen lässt. Der Akteur ist ein Principal, der seine Arbeit macht.

Ich würde erwarten, dass diese Klasse in den meisten Teams die häufigste ist, aus einem strukturellen und nicht aus einem statistischen Grund. Missbrauch braucht eine Person mit einem Motiv, und Kompromittierung braucht einen Angreifer mit einer Gelegenheit, aber ein Fehler braucht nur eine Person mit einer Aufgabe. Ich habe keine Zahl dafür, um wie viel häu-

figer Fehler sind, und ich würde keiner vertrauen, die mir angeboten wird. Die Schlussfolgerung für das Design gilt in beiden Fällen: Kontrollen gegen Fehler schützen vor der Klasse, die jedes Team hat, ob es den anderen beiden je begegnet oder nicht.

Fehler lassen sich am besten angehen, indem man einschränkt, was eine einzelne Anweisung erreichen kann, denn man kann Menschen nicht bitten, ewig vorsichtig zu sein. Rollen, die Tabellen nicht berühren können, die sie nicht brauchen (Kapitel 3), Timeouts, die eine außer Kontrolle geratene Abfrage stoppen (Kapitel 5), die Analyse jeder Anweisung vor ihrer Ausführung (Kapitel 9), ein eigener Weg für Schreibvorgänge mit Vier-Augen-Prinzip (Kapitel 8 und 10) und ein geprüfter Weg, Daten nachträglich zu reparieren (Kapitel 10), gehören alle zu dieser Klasse.

Das Vier-Augen-Prinzip hilft hier nur unter einer Bedingung: Der Reviewer sieht die Anweisung und ihre erwartete Wirkung. Die Absicht zu genehmigen („Rechnung 1042 korrigieren“) ist eine Kontrolle für die Missbrauchsklasse und lässt die Fehlerklasse unberührt. Die Anweisung und die Zahl der Zeilen zu genehmigen, die sie ändern soll, geht Fehler an.

Missbrauch und Neugier

Missbrauch bedeutet hier, dass ein Principal mit legitimem Zugriff ihn für etwas außerhalb der Aufgabe nutzt: den Datensatz eines Kunden aus Interesse lesen, mehr exportieren, als die Arbeit braucht, die Daten von Kollegen ansehen. Er schließt schlichte Neugier ein, die wahrscheinlich die häufigste Form und am schwersten als Fehlverhalten zu bezeichnen ist. Er schließt auch den seltenen vorsätzlichen Fall ein.

Die Antwort des Designs ist nicht Misstrauen gegenüber dem Team. Ein Team muss keinen bösen Willen annehmen, um zu merken, dass ein dauerhafter Grant das Stöbern möglich macht und keine Spur hinterlässt. Die Kontrollen, die zählen, binden den Zugriff an eine Aufgabe und machen ihn sichtbar: schreibgeschützte Rollen, die Restricted-Spalten vorenthalten, maskierte Views auf Confidential-Daten (Kapitel 12), zeitlich begrenzte Grants (Kapitel 8) und ein Audit-Log, das ein Reviewer tatsächlich liest (Kapitel 14). Ohne Nachweis lässt sich „War dieser Zugriff angemessen?“ für niemanden beantworten, auch nicht für die, die nichts falsch gemacht haben.

Kompromittierung

Kompromittierung bedeutet, dass ein Credential oder eine Maschine in den Händen von jemandem ist, der nicht der vorgesehene Principal ist: ein Passwort, das über eine Konfigurationsdatei abgeflossen ist, ein Laptop, der verloren ging, eine Pipeline-Variable, die von etwas gelesen wird, dem man nicht trauen sollte. Die Verteidigung muss nicht wissen, wie es geschehen ist, um dafür zu entwerfen, dass es geschehen kann. Das Design steuert, was das Credential tun könnte, wie lange und ob es jemand merken würde.

Bei der Kompromittierung zählen die Eigenschaften eines Credentials am meisten. Ein Credential, das geteilt, statisch und uneingeschränkt ist, macht aus dem Verlust eines Laptops den Verlust jeder Tabelle. Ein Credential, das auf eine Rolle begrenzt, kurzlebig und an eine Session ausgegeben ist, macht aus demselben Ereignis ein begrenztes und sichtbares. Kapitel 4 und 7 behandeln, wie Credentials ausgegeben werden; Kapitel 3 behandelt, wie wenig sie tragen sollten; Kapitel 14 behandelt, wie man es bemerkt.

> **Hinweis:** Ein Ereignis einzuordnen ist oft eine Ermessensfrage. Eine Korrektur von einer Zeile, die einen ganzen Batch erreicht, ist ein Fehler. Dieselbe Anweisung, absichtlich von jemandem ohne Aufgabe ausgeführt, ist Missbrauch, und mit einem Credential von einem Laptop ausgeführt ist sie Kompromittierung. Die Datenbank sieht jedes Mal dieselbe Anweisung, und das ist das Argument für Logs, die einen Login benennen, und für Kontrollen, die nicht davon abhängen, Absichten zu erraten.

Was geschützt wird

Drei Eigenschaften der Daten sind schützenswert, und das Modell sollte sagen, welche jede Bedrohung berührt. Es sind die bekannten Vertraulichkeit, Integrität und Verfügbarkeit, und jede hat eine Ledgerly-spezifische Bedeutung.

Vertraulichkeit betrifft, wer was lesen kann. Bei Ledgerly ist das Gewicht ungleich verteilt. `iban`, `account_holder_name` und `token_value` sind Restricted, ihre Offenlegung wäre also schwer rückgängig zu machen.

`contact_name`, `contact_email`, `contact_phone` und `billing_address` sind Confidential-PII von Kunden (personenbezogene Daten). Die Beträge in `invoices` und `payments` sind Internal. Eine Bedrohung, die Internal-Daten berührt, verdient weniger Designaufwand als eine, die Restricted-Daten berührt, und das Modell sollte diesen Unterschied zeigen, statt alle Spalten gleich zu behandeln. Datenschutzregime dieser Art fragen tendenziell, welche personenbezogenen Daten Sie halten und wer sie sehen kann, und das ist derselbe Unterschied; die Auslegung liegt bei Ihren Kollegen aus Recht und Compliance.

Integrität betrifft, ob die Daten korrekt bleiben. Bei den meisten Tabellen lässt sich ein Fehler durch einen späteren Schreibvorgang korrigieren, aber `ledger_entries` und `audit_events` sind Append-only, und beim Hauptbuch wiegt Integrität mehr als Geheimhaltung: Ein falscher Eintrag wird durch einen neuen Korrektureintrag über `payments-service` behoben, nie durch eine Änderung. Eine Integritätsbedrohung für diese beiden Tabellen ist daher anders geartet. Sie ist auch die, bei der der Nachweis am wichtigsten ist, denn ein Hauptbuch, das sich unbemerkt ändern lässt, ist ein Hauptbuch, auf das sich niemand verlassen kann.

Verfügbarkeit betrifft, ob die Datenbank die Services bedient, die von ihr abhängen. Es ist die Eigenschaft, die in Sicherheitsdiskussionen am häufigsten vergessen wird, und die, bei der Unfälle überwiegen. Eine lange Sperre, eine Abfrage, die auf dem Primary eine große Tabelle durchsucht, oder eine Flut von Verbindungen aus einer Retry-Schleife kann die Rechnungsstellung und den Zahlungseinzug so gründlich stoppen, wie es ein Angriff könnte. Die Verfügbarkeit bekommt im Modell eigene, gleichrangige Zeilen, verknüpft mit der Verbindungsrichtlinie (Kapitel 5) und damit, wohin Lesezugriffe gehen dürfen (Kapitel 6).

Eine vierte Eigenschaft steht neben diesen dreien: die **Zurechenbarkeit**, also dass die Aufzeichnung sagen kann, welcher Principal was getan hat. Sie ist der Teil Aufzeichnung des Zugriffs. Im Ausgangszustand fehlt sie, denn das Log zeigt nur die gemeinsame Rolle, und dieses Fehlen erhöht die Kosten jeder anderen Bedrohung.

Auswirkungsbereich

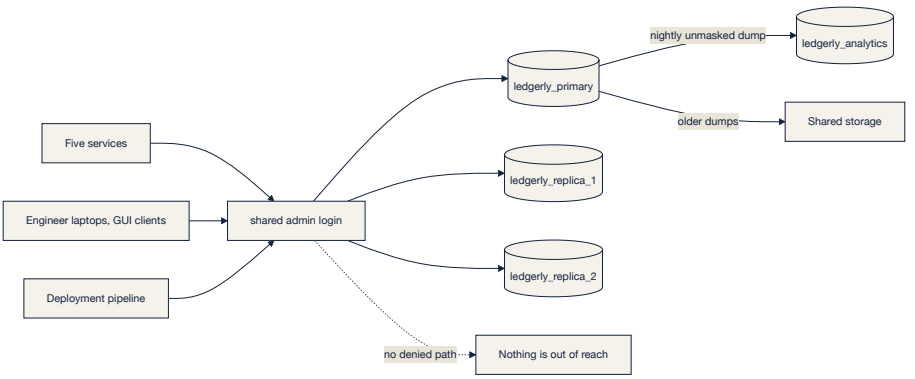
Das Modell braucht ein weiteres Werkzeug, um Bedrohungen zu vergleichen: eine Möglichkeit zu sagen, wie weit sich ein einzelner Ausfall ausbreitet. Der **Auswirkungsbereich** ist, wie viel betroffen sein kann, wenn ein Principal, ein Credential oder eine Handlung schiefgeht. Der Begriff ist nur nützlich, wenn man ihn für etwas Bestimmtes verwendet, also lautet die Frage immer „Was ist der Auswirkungsbereich dieses Logins?“ und nie „Was ist unser Auswirkungsbereich?“

Ein Auswirkungsbereich hat sieben Dimensionen, und es hilft, jede einzeln zu betrachten:

- **Datenreichweite:** welche Tabellen, Spalten und Zeilen der Principal berühren kann.
- **Aktionsreichweite:** ob er lesen, schreiben, löschen oder das Schema ändern kann.
- **Umgebungsreichweite:** mit welchen Datenbanken er sich verbinden kann: dem Primary, den Replikaten, der Analytics-Kopie.
- **Zeitreichweite:** wie lange der Zugriff gültig bleibt, von einer Session bis für immer.
- **Kopienreichweite:** wohin die Daten, die er lesen kann, sonst noch dupliziert wurden und wer diese Kopien erreichen kann.
- **Anweisungsreichweite:** wie viel eine einzelne Anweisung ändern kann, bevor etwas sie stoppt: die Zeilen, die sie berührt, ob sie in einer Transaktion läuft, die zurückgerollt werden kann, und ob ein Timeout sie begrenzt.
- **Netzwerkherkunft:** welche Netzwerke und Maschinen das Credential überhaupt vorzeigen können.

Der gemeinsame Admin-Login im Ausgangszustand von Ledgerly reizt jede Dimension voll aus. Er erreicht alle Tabellen und alle Spalten, er kann lesen, schreiben, löschen und ändern, er verbindet sich mit dem Primary und beiden Replikaten, er läuft nie ab, und seine Daten haben bereits Kopien außerhalb des Primary. Jede Anweisung ist unbegrenzt und wird sofort mit einem Commit abgeschlossen, und jeder Laptop im VPN kann das

Credential vorzeigen. Das Diagramm zeigt, worauf man achten muss: Alles links hält dasselbe Credential, also liegt alles rechts innerhalb eines Bereichs.



Vergleichen Sie es mit der Rolle `app_reporting_job` im Zielzustand: schreibgeschützt, auf einem Replikat, ohne Zugriff auf Restricted-Spalten. Ein Defekt, der im Ausgangszustand Restricted-Daten offenlegen könnte, ist für diese Rolle eine schwere Abfrage auf einem Replikat. Der Defekt ist identisch; der Auswirkungsbereich hat sich geändert.

Damit der Vergleich ehrlich bleibt, schreiben Sie den Bereich pro Login als Liste von Fragen auf und nicht als Wert, und stellen Sie zwei Logins nebeneinander. Die folgende Skizze ist anschaulich gedacht, eine Review-Hilfe und kein Werkzeug, auf dem das Buch aufbaut. Eine unbekannte Antwort wird als unbekannt festgehalten, denn ein Modell, das Unbekanntes zu „in Ordnung“ abrundet, verbirgt genau die Fälle, für die es da ist.

```
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
```

Der Vergleich ist nützlich, weil er ungleichmäßig ist. Wo die zweite Spalte „unknown“ sagt, hat das Modell eine Frage gefunden, die das Design noch nicht beantwortet hat, und das ist ein Befund und keine Lücke in der Tabelle.

Es gelten zwei Grenzen. Ein Bereich beschreibt, was ein Login tun könnte, nicht, was er tun wird; er ist also eine Planungsgrenze und keine Prognose. Und er sagt nichts darüber, wie wahrscheinlich die einzelnen Bedrohungen sind. Ein enger Bereich an einer Tür, die den ganzen Tag offen steht, verdient womöglich mehr Aufmerksamkeit als ein weiter Bereich an einer Tür, die sich einmal im Quartal öffnet, weshalb das Modell Bereich und Exponiertheit verbinden muss.

Das Bedrohungsmodell von Ledgerly

Mit Akteuren, Klassen, Eigenschaften und Auswirkungsbereich an Ort und Stelle passt das Modell in eine Tabelle. Die folgende Tabelle deckt Ledgerlys Ausgangszustand ab. Jede Zeile ist ein Akteur und ein Pfad, mit der Bedrohungsklasse, die für dieses Paar am wichtigsten ist, der Auswirkung auf die obigen Eigenschaften, den Kapiteln, die die Kontrolle liefern, und dem **Restrisiko**: was auch nach diesen Kontrollen noch gilt. „Exponiertheit“ ist ein qualitatives Urteil (hoch, mittel oder niedrig) darüber, wie oft die Tür genutzt wird und wie leicht sie sich missbrauchen lässt, und sie wird aufgeschrieben, damit jemand widersprechen kann.

Akteur	Pfad	Hauptklasse	Auswirkung	Exponiertheit	Geregelt in	Restrisiko
Services (alle fünf)	Gemeinsamer Admin-Login aus der Konfiguration jedes Services	Fehler, Kompromittierung	Jede Tabelle, alle Spalten; Verfügbarkeit, wenn die Verbindungen ausgehen	Hoch	Kapitel 3, 4, 5	Ein Defekt hat weiterhin die Reichweite seiner eigenen Rolle
`reporting-job`	Liest bei manchen Runs den Primary	Fehler	Schwere Lesezugriffe konkurrieren mit Schreibvorgängen; Verfügbarkeit des Primary	Mittel	Kapitel 5, 6	Ein schwerer Lesezugriff auf einem Replikat verlangsamt weiterhin andere Leser des Replikats

Akteur	Pfad	Haupt-klasse	Auswirkung	Expo-niert-heit	Ge-re-gelt in	Restrisiko
Entwickler in Rufbereit-schaft	Laptop, GUI-Client, gemein-sames Admin-Passwort, Auto-Commit	Fehler, Kompro-mittie-rung	Schreibvor-gänge ohne Eingren-zung; Inte-grität jeder Tabelle	Hoch	Ka-pitel 7, 8, 10	Eine von Hand getippte An-weisung inner-halb einer ge-währten Rolle kann weiter-hin falsch sein
Support	`admin-console` unter dem gemeinsamen Login	Miss-brauch, Fehler	Stöbern in Confidential-Kundenda-ten; Ände-rungen beim falschen Kunden	Mittel	Ka-pitel 3, 10, 12	Der Support sieht weiterhin die Spalten, die der Fall ei-nes Kunden braucht
Data-Team	Unmaskierte Analytics-Wie-derherstellung und ältere Dumps	Miss-brauch, Kompro-mittie-rung	Vertraulich-keit jeder Re-stricted- und Confidential-Spalte	Hoch	Ka-pitel 12, 13	Maskierte Da-ten können Werte weiter-hin durch Rückschluss preisgeben
Buchhal-tungsdienst-leister	Exportierte Dateien	Fehler, Miss-brauch	Vertraulich-keit der ex-portierten Spalten und ihre Lebensdauer	Mittel	Ka-pitel 12, 13	Der Dienstlei-ster behält die Datei nach der Zustellung
Deployment-Pipeline	Admin-Login als Pipeline-Va-riable gespeichert	Fehler, Kompro-mittie-rung	Schema- und Datenintegri-tät; alle Tabellen	Mittel	Ka-pitel 4, 11	Eine geprüfte Migration läuft weiterhin mit Eigentümer-rechten
Platform- und DBA-Mitarbeiten-de	Jetzt gemeinsamer Admin-Login; `ledgerly_access_admin` im Zieldesign	Fehler, Miss-brauch, Kompro-mittie-rung	Grants, Kon-trollen und Nachweise können ge-ändert werden	Mittel	Ka-pitel 3, 7, 14	Die Rolle exis-tiert, ihr Miss-brauch ist also möglich; es bleibt nur die Erkennung
Control Pla-ne (Asset)	Gateway, Credential-Broker, Audit-Speicher	Fehler, Kompro-mittie-rung	Jede andere Zeile ändert ihre Bedeutung	Mittel	Ka-pitel 4, 7, 14	Die Kompro-mittierung ei-ner Kompo-nente schwächt den Nachweis, den sie hält

Akteur	Pfad	Hauptklasse	Auswirkung	Exponiertheit	Geregelt in	Restrisiko
Jede menschliche Session	Eine von Hand getippte Anweisung innerhalb einer gewährten Rolle	Fehler	Unbegrenzte Anweisung, sofort mit einem Commit abgeschlossen	Hoch	Kapitel 9, 10	Eine Anweisung kann gültig, genehmigt und in der Wirkung trotzdem falsch sein
Jede Netzwerkherkunft	Das Firmen-VPN erreicht den Primary direkt	Kompromittierung	Ein abgeflossenes Credential funktioniert von jedem Laptop im VPN	Mittel	Kapitel 7	Ein von einer erlaubten Herkunft gestohlenes Credential funktioniert dort weiterhin
Angreifer von außen	Jedes abgeflossene Credential aus den obigen Zeilen	Kompromittierung	Erbt den Bereich des Credential-Eigentümers	Niedrig bis mittel	Kapitel 4, 7, 14	Ein innerhalb seiner Lebensdauer gestohlenes Credential funktioniert, bis es abläuft
Jeder Akteur	Kein benannter Login im Datenbank-Log	Alle Klassen	Zurechenbarkeit für jede andere Zeile	Hoch	Kapitel 7, 14	Die Aufzeichnung ist nur so vertrauenswürdig wie der Speicher, der sie hält

Aus der Tabelle folgen vier Beobachtungen, und sie sind der Grund, sie aufzubauen.

Erstens erscheint der gemeinsame Admin-Login in den meisten Zeilen. Die Tabelle zeigt ein Problem mit vielen Gesichtern, und die erste Kontrolle, eine eigene Rolle pro Principal (Kapitel 3), geht einen großen Teil davon auf einmal an. Ein Modell, das die Reihenfolge der Arbeit offenlegt, erfüllt seine Aufgabe.

Zweitens gehören zu den höchsten Exponiertheiten Türen, die nie als Türen gedacht waren: die Analytics-Wiederherstellung und die älteren Dumps. Keine davon ist im mentalen Diagramm irgendjemandes von der Datenbank ein Zugriffspfad. Deshalb behandeln Kapitel 6 und 13 jede Datenkopie als Tür mit einem Verantwortlichen und einem Ablauf.

Drittens ist die Restrisiko-Spalte der ehrliche Teil der Tabelle. Jede Kontrolle lässt etwas übrig: Eine gewährte Rolle kann weiterhin missbraucht werden, eine gültige Anweisung kann weiterhin falsch sein, einer Aufzeichnung kann man nur so weit trauen wie ihrem Speicher. Eine Zeile ohne aufgeschriebenes Restrisiko ist eine Zeile, bei der niemand gefragt hat.

Viertens benennen die Zeilen zur Control Plane, zur menschlichen Session und zur Netzwerkherkunft ein Asset oder einen Pfad statt eines Akteurs, und auch die letzte Zeile benennt keinen einzelnen Akteur. Fehlende Zurechenbarkeit ist eine Eigenschaft des ganzen Systems, und sie erhöht die Kosten jeder anderen Zeile, denn ohne einen benannten Login lässt sich kein Incident eingrenzen. Kapitel 14 schließt diese Zeile. Frühere Kapitel bauen die Identitäten, die das Schließen möglich machen.

> **Warnung:** Ein fertiges Bedrohungsmodell ist eine Karte der Stellen, an denen Ledgerly am schwächsten ist. Bewahren Sie es an einem Ort auf, der dieselben Zugriffsbeschränkungen hat wie die Systeme, die es beschreibt, und fügen Sie es nicht in ein öffentliches Ticket oder ein Wiki mit großem Publikum ein. Ein Dokument, das unmaskierte Kopien und gemeinsame Credentials auflistet, ist nützlich für die, die es pflegen, und für niemanden sonst.

Das Modell aktuell halten

Ein einmal geschriebenes Bedrohungsmodell ist eine Momentaufnahme, und die obige Tabelle ist an dem Tag veraltet, an dem Ledgerly eine Integration hinzufügt oder das Data-Team eine neue Kopie bekommt. Ich würde es wie ein Runbook als lebendes Dokument mit einem benannten Verantwortlichen behandeln, abgelegt in einem Repository mit denselben Leserechten wie die Systeme, die es beschreibt, da die Tabelle Schwächen auflistet. Das hat einen Preis. Die Zugriffskonfiguration liegt meist in einem breiter lesbaren Repository, sodass eine Änderung dort nicht automatisch im Review des Modells auftaucht. Ich würde diesen Preis in Kauf nehmen und die Lücke mit einer Regel schließen: Ein Pull Request, der die Zugriffskonfiguration ändert, verlinkt auf die passende Änderung des Modells oder sagt, warum keine nötig ist.

Manche Ereignisse sollten eine Aktualisierung auslösen, ohne dass sich jemand daran erinnern muss:

- Ein neuer Service, Job, Dienstleister oder ein neues Tool, das Produktionsdaten erreicht.
- Eine neue Datenkopie: ein neues Replikat, ein Export, ein Staging-Refresh, ein Dump für eine einmalige Analyse.
- Eine Änderung darin, wer was erreichen kann: eine Reorganisation, eine neue Rufbereitschaftsrotation, ein auslaufender Vertrag mit einem Dienstleister.
- Ein Incident oder Near Miss, der ein Beleg dafür ist, welche Zeile falsch war.
- Ein Befund in einem Zugriffs-Review (Kapitel 14), der ein Beleg dafür ist, welche Annahme abgedriftet ist.

Jede Zeile sollte genug festhalten, dass eine fremde Person sie infrage stellen kann: den Akteur, den Pfad, die Klasse, die Auswirkung, das Urteil zur Exponiertheit, die Kontrolle und ihren Verantwortlichen, das Datum des letzten Reviews und die Annahme, von der die Zeile abhängt. Das letzte Feld überspringen Teams am häufigsten, und es ist das nützlichste: „Der Export an den Dienstleister enthält nur diese Spalten“ ist eine Behauptung, die sich prüfen lässt, und eine Zeile, die von einer ungeprüften Behauptung abhängt, ist eine Vermutung.

Eine kleine Prüfung kann Zeilen finden, die veraltet sind. Sie entscheidet nichts. Sie weist einen Menschen auf die Zeilen hin, die am wahrscheinlichsten falsch sind, und sie behandelt ein fehlendes Review-Datum als veraltet. Sie nimmt auch die oben aufgeführten Ereignisse als Eingabe: Wenn eine Pipeline einen Login, eine Rolle oder eine Datenkopie anlegt, hält sie ein Ereignis fest, und ein Ereignis, das keine Zeile abdeckt, öffnet einen Review-Punkt, sodass die Aktualisierung nicht davon abhängt, dass sich jemand erinnert.

```
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
```

Die Taktung ist eine Abwägung. Ein vierteljährlicher Durchgang ist etwas, das ein kleines Team durchhalten kann, und ein jährlicher reicht selten aus, aber ich würde das Intervall danach wählen, wie schnell sich die Türen von Ledgerly ändern. Die Ansprechpartner für Sicherheit und Compliance legen die Fragen fest, die das Review stellt, und das Platform-Team beantwortet sie mit Fakten über das System.

Abwägungen und Kosten

Ein Bedrohungsmodell kostet Zeit beim Schreiben und mehr Zeit bei der Pflege. Für ein kleines Team dauert die erste Version plausibel einige Tage sorgfältiger Diskussion, und die Pflege ist ein wiederkehrender Posten, der mit der Feature-Arbeit konkurriert. Wenn niemand es verantwortet, wird der Preis einmal gezahlt und der Nutzen verfällt. In einem sehr kleinen System mit einem Service und einer Datenbank ist die volle Tabelle mehr, als das Problem braucht; eine halbe Seite, die die Türen und den schlimmsten Auswirkungsbereich jeder Tür auflistet, genügt.

Das Modell kann auch falsche Sicherheit erzeugen: Eine aufgeräumte Tabelle suggeriert, dass jede Bedrohung darin steht. Sie listet die Bedrohungen auf, an die jemand gedacht hat, und eine nicht aufgeführte Tür ist für sie unsichtbar. Der beste Schutz sind die obigen Auslöser für Aktualisierungen und die Gewohnheit, nach jedem Incident zu fragen: „Welche Zeile hätte das vorhergesagt, und wenn keine, welche Zeile fehlt?“

Die Urteile zur Exponiertheit sind subjektiv. Zwei Entwickler werden dieselbe Tür unterschiedlich bewerten, und die Meinungsverschiedenheit ist aufschlussreich, aber eine Bewertung suggeriert mehr Präzision, als die Belege hergeben. Ich würde es bei drei Stufen belassen und der Versuchung widerstehen, Werte zu berechnen.

Es gibt auch einen menschlichen Preis, wenn man die Gruppen von Menschen als Akteure auflistet. Die Rufbereitschaft oder den Support als Bedrohungsakteur zu bezeichnen, kann wie ein Vorwurf wirken. Die Antwort liegt in der Rahmung: Das Modell beschreibt Pfade und Rollen, nicht einzelne Personen, und seine Einträge sagen, was das Design erlaubt. Die Teams in der Tabelle erscheinen, weil sie den berechtigtsten Bedarf haben, Daten zu erreichen.

Wann man das volle Modell nicht einsetzt: Wenn derzeit niemand „Wer hat Zugriff?“ beantworten kann, beginnen Sie mit einem Inventar der Logins und Datenkopien, bevor Sie Bedrohungen schreiben, denn ein Modell auf einem ungenauen Inventar beschreibt ein System, das es nicht gibt. Und verwenden Sie es nicht als Ersatz für eine Entscheidung. Es ordnet und erklärt; das Team wählt weiterhin selbst, welche Kontrollen es zuerst baut und welche Risiken es schriftlich akzeptiert.

Häufige Fehler

- **Nur den Angreifer von außen modellieren.** Das passt zum verbreiteten Bild von Sicherheit, und die Kontrollen sind vertraut. Der Preis ist, dass Fehler und Neugier unbehandelt bleiben. Geben Sie jeder der drei Klassen in der Tabelle eigene Zeilen.
- **Kontrollen statt Bedrohungen auflisten.** „Wir haben ein VPN“ hält eine Antwort ohne die Frage fest. Nennen Sie zuerst, was schiefgehen kann, und danach die Kontrolle, damit sich eine Kontrolle ausmustern lässt, wenn ihre Bedrohung nicht mehr zutrifft.
- **Akteure auslassen, die keine Menschen sind.** Pipelines, Jobs und Exporte übersieht man leicht, weil sich niemand als sie anmeldet, und sie halten tendenziell die meiste dauerhafte Macht. Listen Sie jeden Principal und jeden Export auf.
- **Daten als einheitlich sensibel behandeln.** Wenn jede Spalte „kritisch“ ist, gibt das Modell keine Orientierung, wo man anfangen soll.

Verwenden Sie die vier Datenklassen (Restricted, Confidential, Internal, Public), damit das Gewicht einer Bedrohung dem Gewicht der Daten folgt.

- **Die Verfügbarkeit überspringen.** Eine außer Kontrolle geratene Abfrage sieht nicht wie ein Sicherheitsereignis aus, also fällt sie durch Sicherheits-Reviews. Für ein Zahlungsunternehmen kann sie genauso wichtig sein. Geben Sie der Verfügbarkeit eigene Zeilen.
- **Es nur einmal schreiben.** Das Modell ist einige Wochen lang genau und wird dann zur Folklore. Bestimmen Sie einen Verantwortlichen und fügen Sie Auslöser für Aktualisierungen hinzu.

Zentrale Erkenntnisse

- Ein Bedrohungsmodell ist eine kurze strukturierte Liste von Akteuren, Pfaden, Bedrohungsklassen, Auswirkungen und Kontrollen, und seine Aufgabe ist es, einem Team zu ermöglichen, Türen miteinander zu vergleichen.
- Fehler, Missbrauch und Kompromittierung brauchen unterschiedliche Kontrollen, und ein Design, das nur Außenstehende betrachtet, lässt die ersten beiden Klassen unbehandelt; die Genehmigung der Absicht geht Missbrauch an, während die Genehmigung der Anweisung und ihrer erwarteten Wirkung Fehler angeht.
- Jeder Akteur gehört auf die Liste, einschließlich Pipelines, Batch-Jobs, Dienstleister-Exporten und der Platform-Mitarbeitenden, die die Kontrollen betreiben, denn die Principals, als die sich niemand anmeldet, halten oft die meiste dauerhafte Macht, und die Control Plane ist ebenfalls ein Asset.
- Vertraulichkeit, Integrität und Verfügbarkeit brauchen jeweils eigene Zeilen, und die Zurechenbarkeit, die Fähigkeit zu sagen, welcher Principal was getan hat, erhöht die Kosten jeder anderen Bedrohung, wenn sie fehlt.
- Der Auswirkungsbereich bezieht sich immer auf einen bestimmten Login, ein bestimmtes Credential oder eine bestimmte Handlung und wird entlang von sieben Dimensionen beschrieben: Daten, Aktion, Umgebung, Zeit, Kopien, Anweisung und Netzwerkherkunft.

- Im Ausgangszustand von Ledgerly erscheint ein gemeinsamer Admin-Login in fast jeder Zeile, sodass eine eigene Rolle für jeden Principal die Kontrolle ist, die den größten Teil der Tabelle auf einmal verkleinert.
- Das Modell ist nur so gut wie seine Pflege: Geben Sie ihm einen Verantwortlichen, koppeln Sie Aktualisierungen an Ereignisse wie neue Services und neue Datenkopien, und bewahren Sie es dort auf, wo nur die Personen lesen können, die das System pflegen.

Das Modell sagt Ihnen, welche Türen am wichtigsten sind, und in Ledgerlys Fall führen fast alle zu einem einzigen Login zurück. Kapitel 3 beginnt damit, diesen Login auseinanderzunehmen: wer die Principals sind, was eine Rolle halten sollte und wie wenig jeder von ihnen braucht.

Kapitel 3 — Identitäten, Rollen und minimale Rechte

Stellen Sie sich einen Donnerstagabend bei Ledgerly vor. Eine neue Tabelle, `invoice_disputes`, ging am Dienstag live, und der Sicherheitskontakt stellt dem Platform-Team eine kurze, vernünftige Frage: Wer kann sie lesen? Alex beginnt bei den Grants und findet keine, weil sich jeder Aufrufer als `admin` verbindet, der die Tabelle besitzt und alles lesen kann. Dann sucht Alex, wer sie tatsächlich gelesen hat. Das Datenbank-Log enthält Tausende Zeilen zu der Tabelle, alle vom selben Login geschrieben: vom Worker, vom nächtlichen Report, von zwei Entwicklern auf ihren Laptops und vom Support-Tool.

Niemand kann die neue Tabelle zuordnen. Die ehrliche Antwort auf „Wer kann sie lesen?“ lautet „Wer die Datenbank erreichen kann“, und die ehrliche Antwort auf „Wer hat es getan?“ lautet „Jemand“. Niemand hat etwas falsch gemacht. Das Design hat die Datenbank nie gebeten, diese Aufrufer zu unterscheiden oder irgendeinen von ihnen einzuschränken.

Kapitel 1 teilte den Zugriff in fünf Teile: **Identität**, **Grant**, **Pfad**, **Ausführung** und **Aufzeichnung**. Die These des Buches lautet: Der Zugriff auf Produktionsdatenbanken ist kein Problem der Credential-Verteilung; es ist ein Problem der Autorisierung, der Ausführung und der Zurechenbarkeit. Dieses Kapitel dient der Säule Autorisierung. Es behandelt die ersten beiden Teile, wer fragt (Identität) und was dieser Principal tun darf (Grant), und es markiert, wo Grants enden, denn dort setzt der Teil Ausführung an (Kapitel 9).

Es fügt die ersten Schritte des Zielzustands aus dem durchgängigen Beispiel des Buches hinzu. Jede Tabelle gehört `ledgerly_owner`, und niemand verbindet sich als Eigentümer. Jeder Service bekommt seine eigene Anwendungsrolle mit den schmalsten Grants, die funktionieren. Das Kapitel führt außerdem das vollständige Rollenschema ein, den Weg, auf dem

Principals Rollen erreichen, Grants auf Spaltenebene und die Obergrenze für Just-in-Time-Zugriff. Spätere Kapitel bauen auf diesen Definitionen auf und wiederholen sie nicht.

Das Problem und warum es schwierig ist

Eine Datenbank beantwortet bei jeder Anweisung zwei Fragen: Wer ist das, und darf dieser Principal das tun? Ein gemeinsames Konto macht die erste Antwort bedeutungslos, und ein breites Konto macht die zweite Antwort immer zu einem Ja. Beides ist leicht zu erreichen. Ein Login, ein Passwort und volle Rechte sind das Billigste, was an dem Tag funktioniert, an dem ein Team seine erste Datenbank anlegt.

Die Kosten fallen später an, als fehlender Nachweis. Wenn die Personen und Services, die sich den Admin-Login teilen (sagen wir zwanzig Personen, fünf Services, die Deployment-Pipeline und der Export-Schritt), ihn alle nutzen, kann das Datenbank-Log nicht sagen, wer von ihnen eine Anweisung ausgeführt hat. Das Anwendungs-Log beschreibt, was die Anwendung tun wollte, nicht, was bei der Datenbank ankam. Das ist ein Designproblem und kein Disziplinproblem, denn keine noch so große Sorgfalt Einzelner bringt Information zurück, die das Design verworfen hat.

Das naheliegende Mittel ist eine Richtlinie: nur gewähren, was nötig ist. Eine Richtlinie funktioniert hier nicht, weil die Datenbank die breite Wahl zur einfachen macht. Alles zu gewähren braucht eine Anweisung und schlägt nie fehl. Die schmalste funktionierende Menge zu gewähren erfordert Kenntnis jeder Abfrage, die eine Workload ausführt, einschließlich des monatlichen Jobs, an den sich niemand erinnert, und des Retry-Pfads, der zweimal im Jahr läuft. Ein zu schmaler Grant bricht die Produktion im schlechtesten Moment. Ein zu breiter Grant bricht nichts, also bemerkt ihn niemand. Unter dieser Asymmetrie wachsen Privilegien immer nur.

Der Besitz fügt eine zweite Falle hinzu. In den meisten Engines kann der Eigentümer einer Tabelle ihre Struktur ändern, sie löschen und anderen Zugriff gewähren. Verbindet sich ein Service als Eigentümer, weil er beim Start Tabellen anlegt, hält der Service die Macht, das Zugriffsmodell der Datenbank umzuschreiben. Ist eine Person der Eigentümer, hängt das Mo-

dell vom Arbeitsverhältnis dieser Person ab. Eigentum ist ein strukturelles Privileg und sollte weder in etwas liegen, das Traffic verarbeitet, noch in etwas, das eine Laufbahn hat.

Das Mittel ist eine Struktur und keine Anweisung: ein eigener Login pro Principal, eine kleine Menge von Rollen, die Privilegien tragen, und ein Weg, beides im Einklang mit dem zu halten, was das Team beabsichtigt.

Principals, Logins und Rollen

Dieses Kapitel arbeitet an Identität und Grant und berührt den Pfad dort, wo eine Rolle angibt, von wo sie sich verbinden darf. Ein **Principal** ist alles, was sich authentifiziert. In Ledgerlys Welt gibt es vier Arten, die man trennen sollte.

- **Menschen.** Backend-Entwickler in Rufbereitschaft, Mitarbeitende im Support, das Data-Team und das Platform-Team. Sie kommen und gehen, wechseln Teams und arbeiten von verschiedenen Orten.
- **Services.** `billing-api`, `invoice-worker` und `payments-service`. Sie laufen dauerhaft, in vielen Instanzen, und tragen den größten Traffic.
- **Jobs.** `reporting-job` und sein Export-Schritt. Sie laufen nach Zeitplan und enden. Ein Job braucht meist weniger als ein Service, und für kürzere Zeit.
- **Tools.** `admin-console` ist der klarste Fall. Es ist ein Service in der Art, wie es sich verbindet, handelt aber für eine Person, sodass bei jeder Aktion zwei Principals beteiligt sind: das Tool und der Mensch dahinter.

Der externe Buchhaltungsdienstleister verbindet sich nie mit einer Ledgerly-Datenbank; er erhält Dateien, und Kapitel 13 behandelt diesen Pfad.

Identität, Login und Rolle sind drei verschiedene Dinge

Das Buch hält drei Wörter auseinander, weil ihre Vermischung die Quelle der meisten Verwirrung im Zugriffsdesign ist.

Eine **Identität** ist die verifizierte Behauptung, dass ein Principal der ist, der er zu sein vorgibt. Ein **Login** ist das Konto auf Datenbankebene, mit dem sich ein Principal verbindet. Eine **Rolle** ist ein benanntes Bündel von

Privilegien. Ein Principal hat eine Identität, verbindet sich über einen Login, und der Login ist Mitglied einer oder mehrerer Rollen. Der Login erscheint im Datenbank-Log. Die Rolle entscheidet, was der Login tun darf.

Das ergibt dem Modell einen Satz, den man sich merken sollte: Niemand meldet sich als Rolle an. Eine Rolle existiert, um Logins gewährt zu werden und Privilegien zu halten, und sie ist nie eine Person. Zeigt das Log einen Rollennamen, wo eine Person oder eine Workload-Instanz stehen sollte, wurde das Modell irgendwo umgangen.

Für Menschen erzeugt das Access-Gateway (Kapitel 7) pro Session einen Login, der die Identität der Person trägt, und der Login erhält standardmäßig `human_readonly`. Bei Services verbindet sich jede Instanz mit einem Login, der Mitglied ihrer Rolle `app_*` ist. Im Zielzustand ist dieser Login dynamisch und kurzlebig (Kapitel 4). Im Ausgangszustand ist es der gemeinsame Admin-Login. Die Folge für die zentrale Frage ist klar: Das Log nennt einen Login, und ein Login entspricht einer Person oder einer Workload-Instanz, sodass „Wer hat diese Anweisung ausgeführt?“ eine Antwort hat.

Warum ein gemeinsames Konto Nachweise zerstört

Kapitel 1 hat den Fall ausführlich dargelegt; die Kurzform lautet, dass ein gemeinsames Konto eine Anweisung nicht zurechnen kann, für eine einzelne Person nicht widerrufen werden kann und nicht enger gefasst werden kann, als es der breiteste Aufrufer braucht.

Ein Pfad braucht Sorgfalt: `app_admin_console` ist eine Workload, aber ihre Nutzer sind Menschen. Die Konsole gibt bei jeder Aktion die Identität des menschlichen Principals weiter, und die folgende Skizze zeigt, wie.

```
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 })
```

Der Datenbank-Login benennt die Workload, und der Session-Kontext und `audit_events` tragen die Person und die Correlation-ID, sodass das Anwendungs-Log, der Prüfpfad und die Aufzeichnung der Anweisung selbst einen gemeinsamen Schlüssel haben. Kapitel 14 behandelt, wie die Aufzeichnungen verknüpft werden. Der Kontext wird von der Konsole gesetzt, ist also eine Behauptung der Workload und keine von der Datenbank verifizierte Tatsache; der Vorbehalt zum Pooler im Abschnitt über Zeilenregeln gilt auch für ihn.

Eine Rollenhierarchie in vier Schichten

Ich würde das Rollenschema in vier Schichten aufbauen, von der, die niemand benutzt, nach außen zu denen, die Menschen benutzen. Jede Schicht hat eine Aufgabe.

Schicht eins: der Eigentümer

`ledgerly_owner` besitzt jedes Schemaobjekt: Schemas, Tabellen, Views und Sequenzen. Niemand meldet sich damit an. Es existiert, damit das Eigentum weder an einer Person noch an einem Service hängt.

```
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
```

Schicht zwei: der Migrator

`ledgerly_migrator` führt Schemaänderungen aus. Die Deployment-Pipeline nutzt seinen Login, Menschen nicht, und er hält selbst keine Datenprivilegien. Er ist Mitglied von `ledgerly_owner`. Bei manchen Engines gibt diese Mitgliedschaft die Privilegien des Eigentümers ohne weiteren Schritt weiter, aber nicht das Eigentum: Ein neues Objekt gehört der Rolle, die beim Anlegen aktiv ist, sodass die Migration `ledgerly_owner` immer ausdrücklich annimmt.

Der Migrator hält dauerhafte Eigentümerprivilegien, ausgeübt über ein kurzlebiges Credential. Die Privilegien laufen nie ab; nur jedes einzelne Credential tut es. Drei Dinge begrenzen ihn: Das Credential lebt nur in der Pipeline, es ist kurzlebig (Kapitel 4 und 11), und das Pipeline-Verfahren ist der einzige zulässige Weg, es zu nutzen. Schemaänderungen sind möglich, ohne dass eine Person oder ein Service das Eigentum hält, und wenn diese Grenzen halten, entspricht jede Nutzung des Eigentümers einem Pipeline-Run. Kapitel 11 behandelt, wie Migrationen geprüft werden.

Backfills, die zu einer Schemaänderung gehören, laufen innerhalb der Migration als `ledgerly_migrator`. Jede andere Massenänderung an Daten ist eine Datenkorrektur und folgt Kapitel 10. Ich ziehe diese Linie, weil die Macht des Migrators groß ist und er nicht zum Weg für jede unbequeme Änderung werden sollte.

Schicht drei: Anwendungsrollen

Jede Workload bekommt ihre eigene Rolle. Eine Rolle pro Service bedeutet fünf Definitionen, die mit fünf Codebasen im Einklang gehalten werden müssen. Im Gegenzug ist der Auswirkungsbereich jedes Credentials so

groß wie die Aufgabe dieses Services: Fließt das für `invoice-worker` ab, legt es offen, was der Worker tun konnte, und der Worker hat `payment_tokens` nie gebraucht.

Die Rollen sind `app_billing_api`, `app_invoice_worker`, `app_payments_service`, `app_reporting_job` und `app_admin_console`. Das Präfix sagt, wer sie benutzt: `app_` für Workloads, `human_` für Menschen. Rollen tragen weder Personennamen noch einen Umgebungsnamen.

Schicht vier: menschliche Rollen

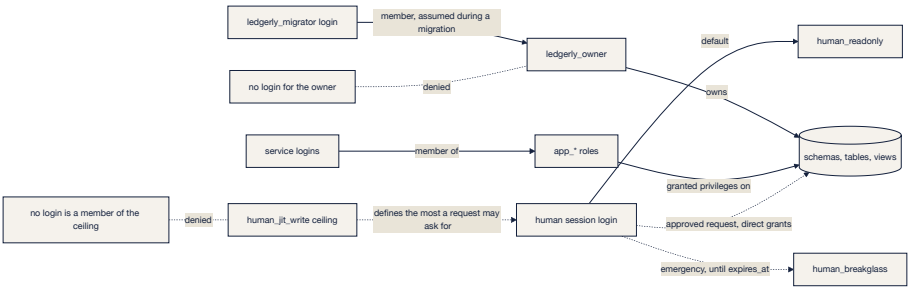
Menschen bekommen drei Rollen, und sie unterscheiden sich darin, wie sie gehalten werden.

- `human_readonly` ist der Standard. Sie liest für Untersuchungen, hält Restricted-Spalten zurück und maskiert Confidential-Spalten dort, wo Kapitel 12 es vorsieht.
- `human_jit_write` ist eine Obergrenze. Sie wird nie einem Login gewährt. Jeder genehmigte Antrag gewährt eine Teilmenge davon, auf benannten Tabellen und benannten Spalten, direkt dem Session-Login und nimmt sie danach wieder weg.
- `human_breakglass` ist breiter Notfallzugriff mit umfangreicher Protokollierung (Kapitel 8 definiert ihn) und verpflichtendem Review im Nachgang.

Die Obergrenze ist die am wenigsten intuitive der drei, und ich komme nach der Mechanik der Grants darauf zurück.

Das Diagramm

Das folgende Diagramm zeigt, wer Mitglied wovon ist und wem was gehört. Achten Sie auf die beiden Dinge, die nicht verbunden sind: Für `ledgerly_owner` existiert kein Login, und kein Login ist Mitglied von `human_jit_write`.



Grants: Schema, Tabelle und Spalte

Ein **Grant** ist die Handlung, und der daraus entstehende Eintrag, einer Rolle ein Privileg zu geben, einem Login oder einer anderen Rolle eine Rolle zu geben oder, nur beim JIT-Zugriff, einem Session-Login direkt ein Privileg zu geben. Ein **Privileg** ist das konkrete Ding, das ein Grant erlaubt, etwa `SELECT` auf einer Tabelle. Die drei Bedeutungen halten das Vokabular präzise, und in diesem Kapitel sage ich, welche ich meine.

Bei null anfangen

Der Ausgangspunkt jeder Rolle ist: gar keine Privilegien. Fügen Sie Grants hinzu, bis die Workload funktioniert, und hören Sie dann auf. Engines unterscheiden sich darin, was ein neuer Login standardmäßig darf: Bei manchen darf jeder Login Objektnamen sehen oder Objekte in einem Standardschema anlegen. Listen Sie auf, was jeder Login automatisch erhält, und entfernen Sie, was eine Rolle nicht braucht.

Privilegien gelten auf drei Ebenen und addieren sich: Eine Rolle hält die Vereinigung von allem, was ihr auf irgendeiner Ebene gewährt wurde, und ein enger Grant nimmt nie einen breiten weg.

- **Schemaebene.** Ob eine Rolle die Objekte in einem Schema überhaupt sehen darf. Bei Engines, die es verlangen, bewirken Grants auf Tabellenebene ohne sie nichts, was den Schemazugriff zu einem nützlichen groben Schalter macht.
- **Tabellenebene.** `SELECT`, `INSERT`, `UPDATE` und `DELETE` auf einer ganzen Tabelle.
- **Spaltenebene.** Dieselben Privilegien nur auf benannten Spalten.

Grants auf Spaltenebene sind wichtig, weil sensible Daten ungleich verteilt sind: `customers` enthält Confidential-Kontaktdaten, `bank_accounts` enthält die Restricted-Spalten `iban` und `account_holder_name`, und `payment_tokens` enthält das Restricted-Feld `token_value`. Ein Grant auf Tabellenebene sagt Ja zu jeder Spalte der Tabelle. Weil sich Privilegien addieren, bedeutet das Einschränken von Spalten, nur auf Spaltenebene zu gewähren und der Rolle auf dieser Tabelle keinen Grant auf Tabellenebene zu geben, weder direkt noch über eine andere Rolle, der sie angehört.

Grants für Anwendungen

Die ersten beiden Anwendungsrollen, auf Tabellenebene:

```
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` liest `ledger_entries` und `payments`, kann sie aber nicht schreiben, denn nur `payments-service` schreibt `payments`, `bank_accounts` und `payment_tokens` und fügt `ledger_entries` ein. Keine der beiden Rollen hält `DELETE` auf irgendetwas; braucht ein Service es, wird der Grant für diese Tabelle mit einer Begründung hinzugefügt, und das Review bemerkt es.

Grants auf Spaltenebene für die anderen Rollen

Rollen mit einem Kreuz in der Matrix sind spaltenbeschränkt. Die folgenden Spaltenlisten sind illustrativ: Welche Confidential-Spalten erscheinen und in welcher maskierten Form, entscheidet Kapitel 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
```

Der letzte Grant gefällt mir am besten. `app_admin_console` darf den `status` einer Rechnung ändern und sonst nichts in der Zeile, sodass der Grant die Absicht ausdrückt und die Datenbank sie durchsetzt, auch wenn sich der Code des Tools später ändert. `app_reporting_job` hält auf keiner Haupttabelle ein Schreibprivileg: Er schreibt seine Ergebnisse nach `ledgerly_analytics`.

> **Hinweis:** Bei Engines mit physischer Replikation werden Rollen einmal für den ganzen Cluster definiert, sodass es eine Regel der Verbindungsschicht (Kapitel 5 und 7) ist, wo sich jede Rolle verbinden darf, und nichts, was die Grants ausdrücken. Kapitel 6 gibt die Einzelheiten.

Die Matrix als Referenz

Die folgende Tabelle ist die Referenz des Zielzustands für die sechs Rollen, die die Haupttabellen lesen oder schreiben. S steht für `SELECT`, I für `INSERT`, U für `UPDATE`, D für `DELETE`, ein Strich bedeutet keinen Zugriff, und ein Kreuz (†) bedeutet spaltenbeschränkt: Restricted-Spalten werden zurückgehalten, und Confidential-Spalten werden dort maskiert, wo Kapitel 12 es vorsieht.

Tabelle	`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	—	—	—

Tabelle	app_billing_api	app_invoice_worker	app_payments_service	app_reporting_job	app_admin_console	human_readonly
payment_tokens	—	—	S, I, U, D	—	—	S†
payments	S	—	S, I, U	S†	S†	S†
invoices	S, I, U	S, I, U	S	S	S, U (nur status)	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

Ich lese die Matrix spaltenweise. Jede Spalte ist die Stellenbeschreibung einer Workload, als Privilegien geschrieben, und ein Reviewer kann fragen, ob sie noch stimmt. Über die ganze Matrix hinweg erreicht allein `payments-service` die `bank_accounts`, nur `payments-service` berührt `payment_tokens` (abgesehen von einem spaltenbeschränkten Lesezugriff durch `human_readonly`), und keine menschliche Rolle schreibt standardmäßig.

Append-only ist strukturell

Keine Anwendungsrolle, keine menschliche Rolle und kein JIT-Grant hält `UPDATE` oder `DELETE` auf `ledger_entries` oder `audit_events`, und das schließt `human_breakglass` ein. Eine Korrektur ist ein neuer Eintrag, nie eine Änderung. Korrekturen am Hauptbuch werden nur über den Korrekturpfad von `payments-service` ausgestellt, nie von einem menschlichen Login.

Es gibt eine strukturelle Ausnahme: `ledgerly_owner`, und damit eine laufende Migration, kann diese Tabellen ändern, weil der Eigentümer die Tabelle ändern oder löschen und sich jedes Privileg darauf selbst gewähren kann. Deshalb prüft Kapitel 11 jede Migration, die sie berührt. Die Kontrolle sitzt im Review, nicht im Privileg.

Defaults, Eigentum und der Superuser

Default-Privilegien

Viele Engines können einen Grant auf später angelegte Tabellen wirken lassen, sei es als Default-Privileg, das an einen Eigentümer gebunden ist, oder als schemaweiter Grant. Das ist anfangs bequem und später eine Falle, weil eine neue Tabelle stillschweigend übernimmt, was der Grant sagt.

Ich bevorzuge einen leeren Default. Eine neue Tabelle ist für niemanden außer dem Eigentümer lesbar, und die Migration, die sie anlegt, nennt jeden Grant in derselben Datei. Der Preis sind ein paar Zeilen mehr pro Migration; der Gewinn ist, dass ein Review die gesamte Zugriffswirkung einer Änderung zeigt.

> **Warnung:** Ein Default-Privileg, das `SELECT` auf künftige Tabellen an eine breite Rolle wie `human_readonly` gewährt, legt die nächste sensible Tabelle in dem Moment offen, in dem sie angelegt wird. Prüfen Sie solche Defaults zuerst und entfernen Sie alle, die künftige Tabellen abdecken.

Die Notation ist illustrativ, weil Default-Privilegien keine neutrale Syntax haben.

```
// 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;
```

Die Gefahr des Eigentums

In gängigen Engines kann der Eigentümer ein Objekt löschen, seine Definition ändern, jedem Privilegien gewähren und in vielen Fällen Zeilenbeschränkungen ignorieren, die für andere gelten. Wer das Eigentum hält, kann das Zugriffsmodell selbst ändern.

Deshalb sind die Regeln um `ledgerly_owner` streng, und die Strenge ist billig. Für ihn existiert kein Login. Anwendungsrollen besitzen nie Tabellen, auch nicht solche, die sie für den eigenen Gebrauch anlegen, denn

„der Service legt seine Tabelle beim Start an“ ist der Weg, auf dem eine Workload einen Teil des Schemas besitzt. Der Eigentümer ist die am wenigsten genutzte Rolle im System, und das ist der richtige Zustand.

Der Superuser

Die meisten Engines haben eine eingebaute administrative Rolle, die Privilegienprüfungen umgeht, auch wenn verwaltete Dienste sie vorenthalten können. Ich nenne sie hier den Superuser; sie liegt außerhalb des Ledgerly-Rollenschemas, weil sie mit der Engine kommt.

Ich würde ihn als versiegeltes Instrument behandeln. Keine Anwendung, kein Job und keine menschliche Session nutzt ihn im Alltag. Sein Credential liegt dort, wo das Platform-Team es abrufen kann und der Abruf einen Eintrag hinterlässt, und er ist für Operationen auf Engine-Ebene gedacht, die keine Rolle ausführen kann, etwa die Reparatur der Replikation. Die ehrliche Grenze ist, dass er jede Kontrolle in diesem Kapitel aufheben kann, und ich kenne kein Design, das diese Macht entfernt, ohne die Wiederherstellung unmöglich zu machen. Machen Sie seine Nutzung selten, sichtbar und geprüft; Kapitel 15 kommt darauf zurück. Routineverwaltung, etwa das Anwenden von Rollendefinitionen oder das Ausstellen von Logins, nutzt stattdessen `ledgerly_access_admin`. Wird der Superuser für Routineaufgaben benutzt, fehlt irgendeiner Rolle ein Privileg, das sie haben sollte.

`ledgerly_access_admin` ist eine Rolle, die von drei Principals genutzt wird, jeweils über einen eigenen Login: den Pipeline-Schritt, der die Zugriffsdefinition anwendet, den Credential-Broker (Kapitel 4) und das Access-Gateway (Kapitel 7). Sie legt Logins an und verwaltet Mitgliedschaften, besitzt keine Tabellen und hält keine Datenprivilegien, und das ist trotzdem weitreichende Macht: Wer sie hält, kann innerhalb der Rollen, die sie verwaltet, alles gewähren. Getrennte Logins halten die drei im Log auseinander, wenn auch nicht in dem, was sie dürfen.

Was die Macht der Control Plane selbst begrenzt, ist daher nicht die Rolle, sondern das Umfeld. Jeder der drei Logins ist nur von der eigenen Komponente aus erreichbar. Jede Aktion geht ins Audit-Log (Kapitel 14), und die Drift-Prüfung weiter unten vergleicht das Ergebnis mit dem Code. Ände-

rungen an der Zugriffsdefinition laufen über einen eigenen kleinen Kreis von Reviewern. Diese Grenzen machen Missbrauch sichtbar; sie machen ihn nicht unmöglich, und ich würde das im Zugriffs-Review so sagen.

Weiter eingrenzen: Views, Zeilenregeln und Mandanten

Views

Ein **View** ist eine gespeicherte Abfrage, die sich wie eine Tabelle verhält. Eine Rolle, der der Zugriff auf den View gewährt wurde, liest, was die Abfrage freigibt, ohne ein Privileg auf der zugrunde liegenden Tabelle zu halten.

```
// 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;
```

> **Hinweis:** In vielen Engines läuft ein View mit den Privilegien seines Eigentümers und nicht mit denen des Aufrufers. Das ist es, was die Eingrenzung wirken lässt, und es ist auch der Grund, warum ein View, der zu viel auswählt, es an jede Rolle weitergibt, der er gewährt wurde. Manche Engines lassen auch das andere Verhalten wählen. Prüfen Sie, was Ihre standardmäßig tut, und testen Sie den View mit einer Rolle, die keine Privilegien auf der Basistabelle hält.

Ich bevorzuge Grants auf Spaltenebene, wo die Engine sie unterstützt, weil der Grant neben der Tabelle lebt. Ein View ist besser, wenn die Form der Punkt ist: ein Join, ein berechneter Wert oder eine stabile Schnittstelle über einer sich ändernden Tabelle. Das Buch verwendet keinen View pro Antrag für JIT-Zugriff, aus zwei Gründen: Ein View muss einen Eigentümer haben, sodass das Gateway, das ihn anlegt, Eigentümermacht bräuchte, und ein View pro Antrag bricht das Eigentumsmodell.

Zeilenregeln und Mandanten: eine Entscheidungsregel

Eine Zeilenregel grenzt Zeilen innerhalb einer Zelle der Matrix ein und ändert nicht, was die Zelle sagt. Der Support sieht jeweils einen Kunden, sodass eine Regel wie „`app_admin_console` sieht nur Zeilen in `invoices`, bei denen `customer_id` dem Kunden des Falls entspricht“ passt, wo ein Spaltenfilter nicht passt. Engines sind sich in der Syntax uneinig, daher zeigt das Buch keine Notation.

Ledgerlys Kunden sind Unternehmen, und jedes erwartet, die Rechnungen eines anderen nicht zu sehen. Diese Isolation ist überwiegend Aufgabe der Anwendung: `billing-api` filtert bei jeder Abfrage nach Kunde, und keine Rolle trennt einen Mandanten vom anderen, weil die Anwendungsrolle alle lesen muss, um irgendeinen zu bedienen.

Verwenden Sie eine Zeilenregel, für den Support oder als zweite Mandantenschicht, nur dann, wenn zwei Bedingungen gelten: Eine falsche Zeile wiegt schwer, und die Anwendung setzt den Kontextwert für jede Session zuverlässig. Der Preis ist, dass jede Anweisung das Prädikat trägt und etwas den Kontext pro Session setzen muss. Hinter einem Pooler kann ein Wert, der auf einer Session gesetzt wurde, den nächsten Entleiher derselben Verbindung erreichen (Kapitel 5 kommt darauf zurück). Fehlt der Kontext, muss die Regel keine Zeilen liefern, und der erste Test ist dieser Fall. Eine Regel, die offen versagt, ist schlimmer als keine. Getrennte Datenbanken sind eine stärkere Grenze zu einem viel höheren Betriebspreis; welche passt, hängt davon ab, wie viele Mandanten Sie haben und was Sie ihnen versprochen haben.

Die Obergrenze für Just-in-Time-Zugriff

`human_jit_write` definiert die Obergrenze: das Meiste, was ein Just-in-Time-Antrag (JIT) verlangen darf, ausgedrückt als Privilegien auf benannten Tabellen und Spalten. JIT-Zugriff ist Zugriff, der für eine bestimmte Aufgabe und eine begrenzte Zeit gewährt wird; Kapitel 8 behandelt Genehmigungen und Laufzeiten, und dieses Kapitel definiert nur die Obergrenze.

Die Rolle wird nie einem Login gewährt. Ein Antrag ist eine Liste von (Privileg, Tabelle, Spalten)-Tripeln, die in die Obergrenze passt und mit dem Antrag gespeichert wird. Bei Genehmigung werden genau diese Tripel di-

rekt dem Session-Login des Antragstellers gewährt, und bei Ablauf werden genau diese Tripel widerrufen. Eine gemeinsame Rolle lässt sich nicht pro Antrag eingrenzen, sodass der Umfang pro Antrag in den Grants liegt und die Rolle den äußeren Rand zieht.

```
// 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
```

Lesen Sie den Block danach, was er auslässt. `bank_accounts` und `payment_tokens` sind nur für die wenigen benannten Nicht-Restricted-Spalten offen und nie für Zeilenbezeichner oder Zeitstempel: `iban`, `account_holder_name` und `token_value` können nie in einem JIT-Antrag erscheinen. `INSERT` ist auf die ersten vier Tabellen beschränkt und kann weder `id` noch `created_at` nennen, sodass eine wiederhergestellte oder neue Zeile immer von der Datenbank vergebene Werte erhält. Die beiden Append-only-Tabellen und die Datensätze der Zugriffsanträge, die dem Gateway gehören, liegen vollständig außerhalb der Obergrenze. Jedes Tripel außer einem `DELETE` nennt seine Spalten.

Zwei Kosten werden hier einmal festgehalten. Spalten, die eine Zeile mit ihrem Eigentümer verknüpfen, etwa `customer_id`, sind nicht Restricted, sodass ein `UPDATE` per JIT sie ändern kann. Eine solche Änderung der Eigentümerverknüpfung bleibt innerhalb der Obergrenze, gehört aber zu einer Klasse gefährlicher Operationen: Die Ausführungsrichtlinie sperrt sie, und sie wird auf den Produktpfad (Kapitel 10) geleitet und nicht als rohes SQL ausgeführt. Das Gate und der Produktpfad sind die Kontrolle, nicht allein die Genehmigung nach dem Vier-Augen-Prinzip. Und ein `SELECT` per JIT zeigt jede Confidential-Spalte, die er nennt, unmaskiert, was Teil dessen ist, was der Genehmiger akzeptiert.

Was die Obergrenze ehrlich hält, ist eine Prüfung und kein Vertrauen. Das Gateway lehnt jeden Antrag außerhalb davon ab (Kapitel 8 zeigt die Prüfung), und der Drift-Test weiter unten stellt sicher, dass kein Login Mitglied von `human_jit_write` ist, den Fehler, der aus einer Obergrenze einen Dauer-Grant machen würde.

Wo die Engine keine Grants auf Spaltenebene hat

Nicht jede Engine unterstützt Grants auf Spaltenebene. Das Buch verwendet genau einen Ausweg: Wo die Engine nicht auf Spalten gewähren kann, wandern die Restricted-Spalten in eigene Tabellen, `bank_account_secrets` und `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
```

Mit dem Ausweg enthalten die Haupttabellen keine Restricted-Spalte, sodass ein JIT-Grant auf Tabellenebene darauf keine Restricted-Daten erreichen kann. Der Preis: Die Details des Antrags sinken auf Tabellenebene, und der Genehmiger hat jede Nicht-Restricted-Spalte der Tabelle genehmigt, Confidential-Spalten eingeschlossen, nicht nur die, die der Antragsteller genannt hat.

Break-Glass und die Operationen oberhalb des Schemas

`human_breakglass` ist die einzige menschliche Rolle, die Restricted-Spalten liest. Sie liest jede Tabelle und schreibt `INSERT` und `UPDATE` auf jeder Tabelle außer den beiden Append-only-Tabellen; auf `bank_accounts` und `payment_tokens` schreibt sie nur `UPDATE`, und nur auf Nicht-Restricted-Spalten. Sie ist für Notfälle gedacht, nicht für Routinekorrekturen, und fehlt in der Matrix, weil nur eine Session sie für ein kurzes Fenster hält; keine dauerhafte Workload tut es.

Restricted-Spalten werden nur über einen Anwendungspfad geändert. Ein Mensch schreibt sie nie mit rohem SQL, weder in einer JIT-Session noch sonst: `admin-console` bittet `payments-service`, den einzigen Schreiber, nachdem eine zweite Person genehmigt hat, und die Änderung wird in `audit_events` festgehalten. Kapitel 10 beschreibt diesen Pfad. Diese Tatsachen markieren den Rand dessen, was Grants je leisten werden. Ein Grant sagt, welche Tabellen und Spalten eine Session berühren darf; er kann nicht sagen, welche Anweisung, wie viele Zeilen oder ob genau diese Änderung die genehmigte ist. Das ist die Aufgabe des Teils Ausführung, und Kapitel 9 behandelt, wie das Gateway jede Anweisung prüft. Zu wissen, wo Grants enden, hilft zu entscheiden, wohin eine neue Fähigkeit gehört.

Rollen als Code und Drift

Ein Rollenschema, das nur in der Datenbank existiert, kann niemand prüfen. Die obigen Grants sind kurz genug, um gelesen zu werden, und sie sollten in der Versionskontrolle liegen, per Review geändert und von der Pipeline angewendet werden, wie das Schema, das sie schützen.

Wo die Definitionen liegen

Ich würde zwei Arten von Dateien führen. Migrationen legen Tabellen an und nennen die Grants jeder Tabelle. Eine getrennte, geprüfte Zugriffsdefinition listet die Rollen, die Mitgliedschaften zwischen Rollen und die Obergrenze auf. Die Pipeline wendet beide in getrennten Schritten an. Die Zugriffsdefinition wird über den eigenen Login des Pipeline-Schritts in `ledgerly_access_admin` angewendet, derselben Rolle, die der Broker

und das Gateway über ihre eigenen Logins nutzen; sie hält keine Datenprivilegien. Rollen anzulegen ist eine stärkere Macht, als Tabellen zu ändern, daher sollte sein Schritt einen eigenen kleinen Kreis von Reviewern haben. Die Frage des Reviewers lautet nicht, ob das SQL gültig ist, sondern ob dieser Grant der schmalste ist, mit dem die Änderung funktioniert. Ein Reviewer mit geöffneter Matrix kann das in einer Minute beantworten.

Ein bestehendes System eingrenzen

Ledgerly beginnt mit einem gemeinsamen Login, dem alles gehört. Ich würde von dort nicht in einer einzigen Änderung zur Matrix wechseln, weil ein zu schmaler Grant die Produktion bricht und das Team nicht wissen kann, was es übersehen hat, bis es fehlschlägt. Ein Weg in vier Schritten:

1. **Geben Sie zuerst jeder Workload einen eigenen Login, mit denselben Privilegien wie zuvor.** Nichts bricht, und das Log beginnt sofort, die Workload zu nennen.
2. **Beobachten.** Nutzen Sie die unter jedem Login aufgezeichneten Anweisungen, um aufzulisten, was die Workload tatsächlich berührt, über einen Zeitraum, der lang genug ist, den langsamen monatlichen Job einzuschließen. Ein Anweisungs-Log kann Literalwerte enthalten, behandeln Sie es daher als sensibel: Es kann Kundendaten oder Restricted-Werte enthalten, die in eine Abfrage kopiert wurden. Schränken Sie ein, wer es liest, legen Sie seine Aufbewahrung fest und protokollieren Sie bevorzugt die Form von Anweisungen mit Parametern, wo die Engine es anbietet.
3. **Definieren Sie die enge Rolle und verschieben Sie den Login hinein**, zuerst in einer Staging-Umgebung, die dieselben Jobs ausführt, dann in der Produktion, wobei die alten Privilegien noch kurze Zeit bereitgehalten werden.
4. **Entfernen Sie die alten Privilegien** und achten Sie auf abgelehnte Anweisungen.

Der Preis ist Kalenderzeit, nicht Aufwand. Teams neigen dazu, den zweiten Schritt zu überspringen, und das vorhersehbare Ergebnis ist ein selten genutzter Pfad, der Monate später bricht.

Drift

Drift ist die Lücke zwischen dem Zugriff, den der Code deklariert, und dem Zugriff, den die Datenbank hält. Sie entsteht, wenn ein Problem nachts mit einem handgeschriebenen `GRANT` behoben wird: Der Grant funktioniert, der Incident endet und niemand schreibt ihn auf. Die Prüfung vergleicht deklariert mit tatsächlich nach Zeitplan und muss laut fehlschlagen, wenn sie eine Seite nicht lesen kann.

```
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)` ist `human_readonly` für einen Session-Login des Gateways, zusätzlich `human_breakglass`, solange ein Break-Glass-Antrag offen ist, und die eine Rolle `app_*`, für die der Broker den Login ausgestellt hat. Die Prüfung fängt daher ein handgeschriebenes `GRANT app_payments_service TO <session login>` oder `GRANT ledgerly_owner TO <login>` ebenso ab wie einen direkten Privileg-Grant.

Hier umfasst `roleGrants` nur Mitgliedschaften von Rolle zu Rolle und Grants von Privileg an Rolle, die der Code deklariert. Logins stehen nicht im Code: Sie kommen und gehen mit Sessions. Menschliche Logins werden gegen die Aufzeichnungen des Gateways geprüft. Die direkten Grants jedes Logins, auch der Service-Logins, werden mit den Tripeln in `access_request_privilege` verglichen, die nicht abgelaufen sind. Ein Service-Login hat keine Anträge, sodass jeder direkte Grant auf einem solchen markiert wird; so fällt ein nachts handgeschriebenes `GRANT` eines Privilegs an einen Service-Login auf. Ein Grant, der seinen Antrag überdauert hat, oder ein Break-Glass-Mitglied ohne offenen Antrag wird markiert, auch wenn der Revoke-Job Erfolg meldet. JIT-Zugriff funktioniert über direkte Grants und den Ablauf über einen Revoke-Job; stoppt der Job, bleiben die Grants bestehen, und diese Prüfung bemerkt es unabhängig vom Job (Kapitel 8 behandelt die Paarung). Die Prüfung liest nur Grants und Mitgliedschaften. Sie sieht nicht, was eine Session damit tut, was wieder der Teil Ausführung ist.

Die Prüfung repariert nichts, denn ein automatisches Widerrufen unbekannter Grants in der Produktion ist eine neue Art, die Produktion lahmzulegen. Sie sagt auch nicht, ob der deklarierte Zustand richtig ist: Ein falscher Grant, korrekt deklariert, besteht sie. Dieses Urteil gehört zum Zugriffs-Review aus Kapitel 14.

Abwägungen und Kosten

Jede Kontrolle in diesem Kapitel hat einen Preis, und den Preis zahlen bestimmte Menschen.

Rollen pro Workload kosten Wartung. Fünf Rollen bedeuten fünf Definitionen, die mit fünf Codebasen im Einklang gehalten werden müssen. Braucht ein Service eine neue Tabelle, aktualisiert jemand die Rolle in der-

selben Änderung; landet der Grant nach dem Code, schlägt das Deployment fehl. Die Gegenmaßnahme ist, den Grant zum Teil der Migration zu machen. Bei einem Service und zwei Entwicklern ist dieser Aufwand nicht nichts, und ein kleineres Schema aus einem Eigentümer, einer Anwendungsrolle und einer schreibgeschützten Rolle ist ein vernünftiger erster Schritt.

Enge Grants führen zu abgelehnten Anweisungen. Manche Ablehnungen sind Fehler in den Grants, und manche sind Code, der weiter gegriffen hat, als er sollte. Wer in der Rufbereitschaft ist, zahlt für beides, wenn während eines Incidents eine Ablehnung auftaucht. Staging mit denselben Rollen wie die Produktion zu betreiben, verringert die Überraschung; die Kosten zu ignorieren führt dazu, dass Teams einen Grant in Eile erweitern, und die Eile ist der Anfang von Drift.

Grants auf Spaltenebene erhöhen die Review-Last. Jede Spaltenliste muss gepflegt werden, wenn sich die Tabelle ändert. Eine neue sensible Spalte ist unter einem Grant auf Tabellenebene standardmäßig offengelegt, daher gewähre ich bei sensiblen Tabellen auf benannten Spalten und nehme die zusätzlichen Zeilen in Kauf. Wo der Engine die Funktion fehlt, kostet der Ausweg Joins, zusätzliche Tabellen für Migrationen und eine gröbere JIT-Genehmigung.

Views und Zeilenregeln fügen bewegliche Teile hinzu. Sie sind ihr Gewicht wert, wenn die Form oder die Isolation der Punkt ist, und ein schlechter Handel, wenn sie hinzugefügt werden, um sorgfältig zu wirken. Eine Zeilenregel, deren Kontext fehlen kann, ist Sicherheitstheater: Sie fügt jeder Abfrage Reibung hinzu und versagt offen in dem Fall, für den sie geschrieben wurde.

Disziplin beim Eigentum bremst manche Menschen. Entwickler, die gewohnt sind, eine Tabelle in einer Konsole anzulegen, müssen nun den Weg über eine Migration gehen. Die Reibung ist beabsichtigt, und sie ist auf einer Wegwerf-Entwicklungsdatenbank weniger wert als in der Produktion. Wenden Sie das Modell zuerst auf die Produktion an und lockern Sie es anderswo nur, wenn Sie sagen können, was Sie aufgeben.

Wann man nicht so viel Struktur einsetzt. Ein Prototyp mit einem Entwickler und ohne Kundendaten braucht nicht das volle Rollenschema. Die Aufteilung in Eigentümer, Anwendung und schreibgeschützt genügt bis

zum zweiten Service oder zur ersten sensiblen Spalte. Lassen Sie das Schema wachsen, wenn „Wer hat das ausgeführt?“ aus dem Log nicht beantwortet werden kann oder wenn das Credential einer Workload mehr Schaden anrichten könnte, als ihre Aufgabe rechtfertigt.

Häufige Fehler

- **Die Anwendungsrolle als Eigentümer behandeln.** Das ist verlockend, weil Migrationen dann beim Start mit einem Schritt weniger laufen. Der Service hält dann die Macht, zu löschen und neu zu gewähren. Verwenden Sie einen getrennten Migrator und einen Eigentümer ohne Login.
- **Gewähren, um die Abfragen des letzten Jahres abzubilden.** Eine vom breitesten bestehenden Login kopierte Rolle behält jedes Privileg, das jemals bequem war. Leiten Sie die Rolle von dem ab, was die Workload jetzt tut, beobachtet unter ihrem eigenen Login.
- **Eine Rolle pro Person anlegen.** Rollen sind Bündel von Privilegien, und Logins sind pro Person. Eine nach einer Einzelperson benannte Rolle macht das Schema zu einer Mitarbeiterliste. Halten Sie Rollen bei Aufgaben.
- **Eine Rolle für mehrere Services nutzen, weil sie „ähnlich“ sind.** Die Services driften auseinander, und die Rolle behält die Vereinigung dessen, was jeder braucht. Eine Rolle pro Service kostet ein paar Zeilen.
- **Defaults für künftige Tabellen belassen.** Ein Default-Privileg spart eine Zeile in jeder Migration und legt die nächste sensible Tabelle einer breiten Rolle offen. Halten Sie den Default leer und gewähren Sie ausdrücklich.
- **Die Obergrenzen-Rolle für eine arbeitsreiche Woche einem Login gewähren.** Die Obergrenze wird dann zu einem Dauer-Grant mit einem freundlichen Namen. Halten Sie die Obergrenze ungebunden: Nur die Antragsprüfung liest sie, und der Drift-Test markiert jede Mitgliedschaft.
- **Eine sensible Spalte hinter einem View verstecken und die Tabelle lesbar lassen.** Der View grenzt nichts ein für eine Rolle, die die Basistabelle erreichen kann. Widerrufen Sie auf der Tabelle und testen Sie mit einer Rolle, die kein Tabellenprivileg hält.

- **Einem deklarierten Schema vertrauen, das niemand vergleicht.** Rollen in der Versionskontrolle halten das `GRANT` um 2 Uhr nachts nicht auf. Vergleichen Sie deklariert und tatsächlich nach Zeitplan und lassen Sie den Vergleich laut fehlschlagen, wenn er nicht lesen kann.
- **Den Superuser für Routinearbeit verwenden.** Er funktioniert jedes Mal, deshalb breitet er sich aus. Braucht eine Routineaufgabe ihn, fügen Sie das fehlende Privileg der richtigen Rolle hinzu.

Zentrale Erkenntnisse

- Ein gemeinsamer Login zerstört Nachweis und Umfang zugleich: Das Log kann nicht sagen, wer gehandelt hat, und der Login muss alles dürfen, was sein breitester Aufrufer tut.
- Halten Sie Identität, Login und Rolle auseinander: Ein Principal hat eine Identität, verbindet sich über einen Login, und der Login ist Mitglied von Rollen; niemand meldet sich als Rolle an.
- `ledgerly_owner` besitzt alles und hat keinen Login, `ledgerly_migrator` nimmt ihn nur während einer Migration an, und jede Workload bekommt ihre eigene Rolle `app_*`, zugeschnitten auf ihre Aufgabe.
- Beginnen Sie jede Rolle ohne Privilegien, nutzen Sie Grants auf Spaltenebene, wo sensible Spalten neben gewöhnlichen liegen, halten Sie Default-Privilegien leer und weichen Sie nur dort auf getrennte Secrets-Tabellen aus, wo die Engine nicht auf Spalten gewähren kann.
- `human_jit_write` ist eine Obergrenze, die nie einem Login gewährt wird: Jeder genehmigte Antrag gewährt eine Teilmenge direkt dem Session-Login, und kein Antrag kann eine Restricted-Spalte enthalten.
- Rollen und Grants gehören in die Versionskontrolle und in geprüfte Pipeline-Schritte, und eine geplante Drift-Prüfung, die laut fehlschlägt, wenn sie die Datenbank nicht lesen kann, hält den deklarierten und den tatsächlichen Zustand zusammen.
- Jede Schicht kostet jemanden etwas: Wartung bei Rollen, abgelehnte Anweisungen bei engen Grants, Review-Last bei Spaltenlisten und eine gröbere Genehmigung, wo der Engine Grants auf Spaltenebene fehlen.

Rollen sagen, was ein Principal tun darf, sobald er verbunden ist, aber sie sagen noch nichts darüber, wie ein Service beweist, wer er ist. Kapitel 4 geht den nächsten Schritt: wie Credentials zu Workloads gelangen, warum kurzlebige und dynamische Credentials die Gestalt des Problems verändern und was das Passwort in der Konfigurationsdatei ersetzt.

TEIL II

Zugriff durch Anwendungen

Kapitel 4 — Credentials und Secrets

Stellen Sie sich einen Freitagnachmittag bei Ledgerly vor. Jemand aus dem Backend-Team ist in eine andere Gruppe gewechselt, und die Ansprechperson für Sicherheit fragt höflich, ob man das gemeinsame Admin-Passwort ändern könnte. Alex vom Platform-Team stimmt zu, dass es geändert werden sollte. Dann beginnt Alex aufzulisten, wo das Passwort überall liegt: in der Konfiguration aller fünf Services, in den Variablen der Pipeline, im Export-Schritt von `reporting-job`, in einem Team-Passwortmanager und in den gespeicherten Verbindungen mehrerer Desktop-Clients.

Die Änderung in der Datenbank dauert eine Sekunde. Sie überall im selben Moment durchzuführen bedeutet einen koordinierten Neustart jedes Services während der Geschäftszeiten, und Robin, dieses Wochenende in Rufbereitschaft, stellt die berechtigte Frage: Was geschieht mit den Rechnungen, die gerade in Verarbeitung sind? Die Änderung wird auf „ein ruhiges Zeitfenster“ verschoben, das nie kommt. Das Passwort überdauert die Person und dann die nächste Person.

Dieses Kapitel beantwortet die Frage, die dieser Freitag aufgeworfen hat: Wie kommt ein Service an ein Credential, und wie ändert man eines ohne Ausfall? Es fügt dem laufenden Beispiel den nächsten Schritt des Zielzustands hinzu. Credentials für Services werden kurzlebig oder dynamisch, und sie erreichen Workloads über die Identität statt über eine gemeinsame Datei.

Credentials gehören zum Teil Identität des Fünf-Teile-Modells; sie sind notwendig, aber nicht die Antwort. Die These des Buches lautet: „Der Zugriff auf Produktionsdatenbanken ist kein Problem der Credential-Verteilung; es ist ein Problem der Autorisierung, der Ausführung und der Zurechenbarkeit.“ Ein besserer Umgang mit Credentials verkleinert den Kreis

derer, die einen Login erhalten können. Er sagt nichts darüber, was ein Login tun darf, sobald er drin ist, oder was er getan hat, und das sind die Aufgaben der Kapitel zu Grants, Ausführung und Aufzeichnungen.

Das Problem und warum es schwierig ist

Ein **Credential** ist das, was ein Principal zur Authentifizierung vorlegt; ein **Secret** ist jeder Wert, der bei Offenlegung Zugriff gewährt. Ein Passwort ist beides. Kapitel 3 hat jedem Service seine eigene Rolle gegeben, sodass ein Login aussagt, was er darf. Dieses Kapitel behandelt die andere Hälfte: wie sich der Login ausweist und wie lange dieser Beleg gültig bleibt.

Ein Passwort hat einen Lebenszyklus, auch wenn ihn niemand entworfen hat. Es wird erzeugt, dorthin kopiert, wo ein Service es lesen kann, monatelang benutzt und schließlich geändert oder vergessen. Jede Kopie ist ein Ort, an dem eine Person oder ein Prozess es lesen kann, und die Frage „Wer kann dieses Passwort sehen?“ hat so viele Antworten, wie es Kopien gibt. Im Ausgangszustand lautet die Antwort: „ungefähr alle im Backend-Team und alles, was die Konfiguration eines Services lesen kann“.

Das naheliegende Mittel ist ein besseres Passwort: länger, zufällig, ordentlicher abgelegt. Das hilft gegen Erraten, und Erraten war nie das Hauptrisiko. Das Risiko ist die Zahl der Orte, an denen der Wert liegt, und die Tatsache, dass sich keiner davon leeren lässt, ohne die anderen anzufassen. Ein starkes Passwort in fünf Konfigurationsdateien bleibt fünf Kopien.

Die zweite Schwierigkeit ist die **Rotation**, also das Ersetzen eines Credentials durch ein neues nach Plan oder bei Verdacht. Rotation ist schwierig, weil ein Credential geteilter Zustand zwischen zwei Systemen ist: Die Datenbank kennt den Wert, und jeder Consumer kennt ihn ebenfalls. Wird zuerst die eine Seite geändert, bricht die andere. Ein Team, das ein Passwort nicht ohne Ausfall ändern kann, wird es vernünftigerweise nicht mehr ändern, und eine Rotationsrichtlinie, die niemand ausführen kann, fügt einen Kalendereintrag hinzu und beseitigt keine Gefährdung.

Was Credentials nicht leisten können

Nehmen Sie einen Login des Reporting-Jobs, eine Stunde gültig, der in einen Chat-Kanal gelangt ist. Nach jedem Maßstab dieses Kapitels ist er ein gutes Credential: Er ist eindeutig, kurzlebig und wird über die Identität

abgerufen. Für seine Stunde kann er dennoch die drei Spalten von `customers` lesen, die der Rolle gewährt sind (`id`, `created_at` und `status`), und zwar in jeder Zeile, denn das darf `app_reporting_job` lesen. Kurze Lebensdauern verkleinern das Zeitfenster; sie verkleinern nicht die Rolle. Drei Dinge begrenzen ein Leck, und keines davon ist das Credential selbst: die **Rolle**, die es trägt (Kapitel 3), die **Netzwerkreichweite**, die entscheidet, von wo aus es benutzt werden kann (Kapitel 7), und das **Log**, das aufzeichnet, was es getan hat (Kapitel 14). Ein Login, der nur aus dem Netz des Poolers vorgelegt werden kann, mit einer Rolle, die keine Restricted-Spalten sehen kann, ist ein kleinerer Incident, als ihn irgendeine Passwortrichtlinie erreichen kann.

Dasselbe gilt für das stärkste Design weiter unten. Ein Broker, ein Lease und die Workload Identity beantworten alle die Frage, wer einen Login erhalten darf. Sie entscheiden nicht, was der Login tun darf, und sie hindern einen erlaubten Login nicht daran, eine schädliche Anweisung auszuführen.

Das ist eine Frage der Technik, nicht der Richtlinie. Eine Regel, die lautet „alle 90 Tage rotieren“, sagt nichts darüber, wie die fünf Services den neuen Wert im selben Augenblick erfahren. Die Antwort liegt darin, wie Credentials ausgestellt, zugestellt und ersetzt werden, und das ist ein Design, das das Team bauen muss.

Der Lebenszyklus eines Credentials

Es hilft, ein Credential von der Entstehung bis zur Ausmusterung zu verfolgen und an jeder Station zu fragen, wer es lesen kann und was es offenlegen würde. Die üblichen Orte sind diese.

Ort	Wer es lesen kann	Wie es üblicherweise durchsickert	Besseres Zuhause
Konfigurationsdatei im Repository	Alle mit Zugriff auf das Repository, für immer, über die Historie	Kopiert in einen Fork, einen Chat, ein Ticket	Ein Secret-Manager, per Name referenziert
Umgebungsvariable	Der Prozess, seine Kindprozesse und alle, die den Prozess oder seine Crash-Dumps untersuchen können	Von einem Debug-Endpoint oder einem Fehlerbericht ausgegeben	Eine beim Start übergebene Datei oder ein beim Start übergebener Speicherbereich, mit engen Dateiberechtigungen

Ort	Wer es lesen kann	Wie es üblicherweise durchsickert	Besseres Zuhause
Container-Image oder Build-Artefakt	Alle, die das Image abrufen können	Beim Build eingebaut, sodass jede Kopie es enthält	Nichts im Image; beim Start abgerufen
CI/CD-Variable und Job-Log	Pipeline-Administratoren; manchmal jeder Job	Von einem Skript wiedergegeben, von einem fehlschlagenden Befehl ausgegeben	Kurzlebiger Wert, der von der eigenen Identität des Jobs abgerufen wird
Team-Passwortmanager	Alle im gemeinsamen Vault	Aus Gewohnheit geteilt; keine Aufzeichnung, wer es gelesen hat	Individueller Zugriff mit Aufzeichnung, für die wenigen, die übrig bleiben
Gespeicherte Verbindung eines Desktop-Clients	Wer diesen Laptop benutzt	Verlorener Laptop oder Laptop in einem Backup	Kein gespeichertes Credential; Session-Credentials von einem Gateway (Kapitel 7)

In der Tabelle fallen zwei Dinge auf. Erstens sind die Lecks fast nie raffiniert. Es sind Kopien, die aus Bequemlichkeit entstehen: ein Wert, der in ein Ticket eingefügt wird, damit ein Kollege ein Problem nachstellen kann, eine Variable, die bei der Fehlersuche an einem Deployment ausgegeben wird. Zweitens ist die Spalte „Besseres Zuhause“ größtenteils immer dieselbe Antwort: Der Wert liegt an einem verwalteten Ort, und die Consumer rufen ihn ab, wenn sie ihn brauchen.

> **Warnung:** Ein Secret, das in einer Image-Schicht, in einer Repository-Historie oder in einem Job-Log aufgetaucht ist, muss als offengelegt gelten, auch wenn es niemand gelesen hat. Das Löschen der Zeile löscht die Kopien nicht. Abhilfe ist, das Credential zu ersetzen; deshalb braucht ein Team einen Rotationspfad, der funktioniert, bevor der Tag kommt, an dem man ihn braucht.

Ein Secret hat noch eine Eigenschaft, die einen Namen verdient: wie lange es gültig bleibt. Angenommen, ein Passwort ist drei Jahre gültig: Es trägt drei Jahre Gefährdung durch jede Kopie, die je gemacht wurde. Eine alte

Kopie eines Credentials, das eine Stunde hält, ist wertlos. Die Zeiträume sind beispielhaft. Deshalb ist die Lebensdauer wichtiger als die Stärke.

Secret-Manager und wie ein Service ein Secret liest

Ein **Secret-Manager** ist ein Service, der Secrets speichert, steuert, wer welches lesen darf, und jedes Lesen aufzeichnet. HashiCorp Vault und die Speicher für Secrets der großen Cloud-Anbieter sind Beispiele. Das Pattern ist wichtiger als das Produkt: Ein Ort hält den Wert, der Zugriff wird pro Principal entschieden, und jedes Lesen hinterlässt eine Aufzeichnung.

Das Passwort in einen Secret-Manager zu verlegen verringert allein noch nicht die Zahl der Leser. Es ändert, wer sie sind, und gibt jedem Lesevorgang einen Namen: Eine bestimmte Service-Identität liest einen bestimmten Pfad, und der Secret-Manager hält fest, wann. Das ist ein echter Gewinn für Untersuchung und Widerruf und ein bescheidener Gewinn für die Gefährdung.

Das erste Secret

Ein Service braucht etwas, womit er sich gegenüber dem Secret-Manager ausweist, bevor er irgendetwas lesen kann. Ist dieses Etwas ein weiteres gespeichertes Passwort, hat sich das Problem nur eine Ebene nach oben verschoben, und nichts ist gelöst. Das ist die Bootstrap-Frage, manchmal auch First-Secret-Problem genannt.

Die Antwort ist die **Workload Identity**: eine Identität, die die Plattform einem laufenden Workload gibt, etwa einer Service-Instanz oder einem Job, damit er sich ohne gespeichertes Passwort authentifizieren kann. Die Plattform, die den Workload startet, bürgt für ihn, und der Secret-Manager vertraut dieser Aussage. Der Service hält für diesen Schritt nie einen eigenen langlebigen Wert. Ich betrachte die Workload Identity als das, was den Rest des Designs möglich macht, weil sie den Regress beendet.

Beim Start lesen und erneut lesen

Ein Service hat zwei vernünftige Zeitpunkte, um sein Datenbank-Credential zu lesen: einmal beim Start oder immer dann, wenn er eine Verbindung öffnet. Einmal zu lesen ist einfach und funktioniert bei statischen Credentials, die sich selten ändern. Es versagt, wenn sich das Credential doch än-

dert, weil der laufende Prozess den alten Wert bis zum Neustart behält. Das Lesen beim Verbindungsaufbau, über einen Client, der den Secret-Manager nach einem Authentifizierungsfehler erneut fragt, kommt ohne Neustart mit einer Rotation zurecht.

```
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
```

Der Block zeigt die beiden Fehler, auf die es ankommt. Ein Authentifizierungsfehler löst ein frisches Lesen aus, sodass aus einer Rotation kein Ausfall wird. Ein nicht erreichbarer Secret-Manager lehnt den Start ab, weil der Rückgriff auf eine Ersatzkopie die Datei wiederherstellt, die dieses Design beseitigt hat. Diese Ablehnung hat ihren Preis, auf den der Abschnitt „Abwägungen und Kosten“ zurückkommt.

> **Hinweis:** Der Secret-Manager selbst ist eine Tür. Wer den Pfad `db/billing-api` lesen darf, wer ihn ändern darf und ob Lesezugriffe im Log festgehalten werden, sind dieselben Fragen wie bei der Datenbank. Jedem Service Lesezugriff auf jeden Pfad zu geben, würde das gemeinsame Passwort mit einem zusätzlichen Schritt wieder aufbauen.

Rotation ohne Ausfall

Das Mittel gegen „die eine Seite zuerst ändern, und die andere bricht“ besteht darin, nie einen Moment zu haben, in dem nur ein Wert funktioniert. Das ist das **Zwei-Credential-Pattern**, oft A/B-Rotation genannt. Jeder Service hat zwei Logins, A und B, beide Mitglieder derselben Rolle `app_*`. Zu jedem Zeitpunkt ist einer aktiv und der andere inaktiv.

Zum Rotieren ändern Sie den inaktiven Login, veröffentlichen den neuen Wert, weisen die Consumer an umzuschalten, warten, bis der alte Login nicht mehr benutzt wird, und machen ihn dann ungültig. Zu keinem Zeitpunkt hält ein Consumer einen Wert, den die Datenbank nicht mehr akzeptiert. In Pseudocode, mit markierten Fehlerstellen:

```
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" })
```

Drei Details tragen das Design. Erstens sind die Datenbank und der Secret-Manager getrennte Systeme ohne gemeinsame Transaktion, sodass die Reihenfolge der Schritte einen Ausfall begrenzt: Der Wert, der gerade aktiv ist, wird nie geändert, bevor sein Ersatz lesbar ist. Ein Fehler in einem beliebigen Schritt lässt das vorherige Credential weiter funktionieren. Zweitens die Abflussprüfung. Die Rotation ist erst abgeschlossen, wenn nichts mehr den alten Login benutzt, und ein Service, der sein Secret nie neu liest, fällt in diesem Moment auf, nicht in einem Ausfall drei Wochen später. Drittens überschreibt der letzte Schritt den alten Login mit einem Wert, den niemand aufbewahrt, aber er läuft erst nach dem Abfluss. Solange irgendeine Session den alten Login hält, funktioniert der alte Wert.

Der Preis ist, dass jeder Service nun zwei Logins und einen Job hat, der beide kennt, und dass der Consumer das Secret nach einem Fehler neu lesen muss. Ein Service, der sein Credential nur einmal beim Start liest, braucht zum Rotieren einen schrittweisen Neustart, und genau dieser Neustart macht aus einer A/B-Rotation wieder einen Kalendertermin. Das gemeinsame Admin-Passwort im Ausgangszustand zu rotieren ist aus diesem Grund schwierig: Es gibt keinen zweiten Login, und kein Consumer kann irgendetwas neu lesen.

Ich würde nach einem Zeitplan rotieren, den das Team durchhalten kann, nicht nach dem kürzesten, den es aufschreiben kann. Ein vermutetes Leck ist ein anderer Weg, beschrieben in „Wenn ein Credential durchsickert“.

Dynamische und kurzlebige Credentials

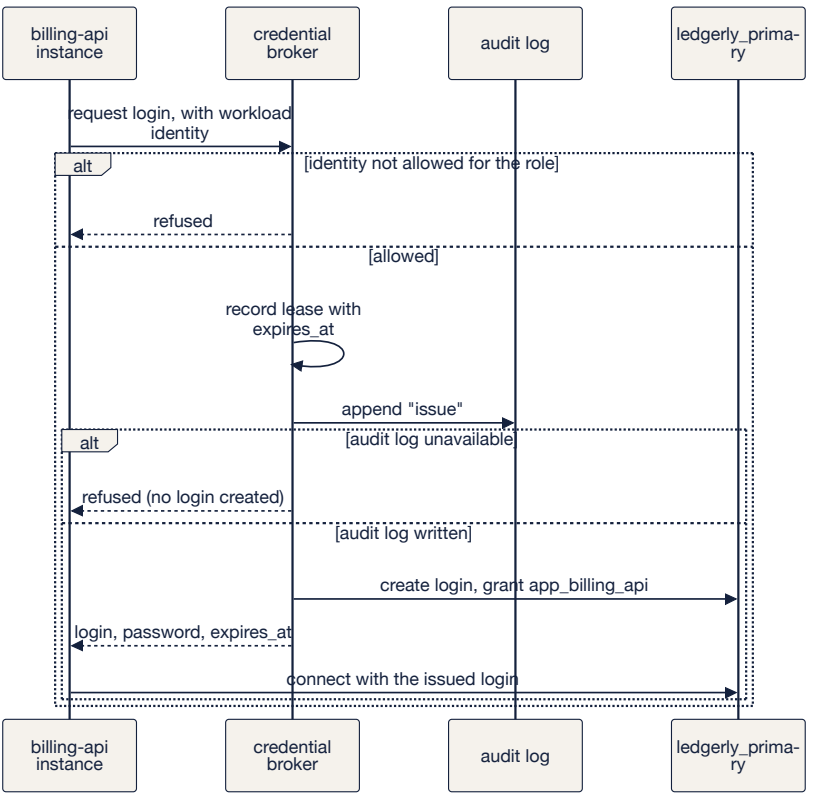
Ein rotiertes statisches Credential hat zwischen den Rotationen noch immer eine lange Lebensdauer. Der nächste Schritt besteht darin, die Lebensdauer so kurz zu machen, dass die Rotation kein Ereignis mehr ist. Ein **kurzlebiges Credential** läuft innerhalb einer kurzen, definierten Zeit von selbst ab. Ein **dynamisches Credential** ist ein kurzlebiges Credential, das auf Anforderung für genau einen Principal erzeugt wird, sodass zwei Anforderungen nie denselben Wert erhalten. Alle dynamischen Credentials sind kurzlebig; nicht jedes kurzlebige Credential ist dynamisch.

Der übliche Mechanismus ist ein **Credential-Broker**: ein Service, der auf Anforderung einen Login in der Datenbank anlegt, ihm die Mitgliedschaft in der passenden Rolle `app_*` gibt und das Credential des Logins zusammen mit einem **Lease** an den anfragenden Workload übergibt. Ein Lease ist ein Datensatz, der festhält, wer das Credential hält, für welche Rolle und bis wann. Läuft das Lease ab oder widerruft jemand es, entfernt der Broker den Login.

Bei Ledgerly verbindet sich der Broker mit einem eigenen Login, einem Mitglied von `ledgerly_access_admin`, der Rolle, die Logins anlegt und löscht und Rollenmitgliedschaften verwaltet. Sie besitzt keine Tabellen, ist nicht der Eigentümer, und niemand nutzt sie für die Arbeit mit Daten. Das ist eine geringere Macht als das Eigentum, und sie ist dennoch groß: Wer die Rolle hält, kann innerhalb der Rollen, die sie verwaltet, alles gewähren, Anwendungs- und menschliche Rollen eingeschlossen. Der Broker

braucht nur einen kleinen Teil dieser Macht. Ihm weniger zu geben würde bedeuten, die Rolle aufzuteilen, was eigene Kosten hat; hier nutzt der Broker die Rolle so, wie sie ist. Die Rolle ist eine einzige Rolle, die drei Principals nutzen, jeder über seinen eigenen Login: den Pipeline-Schritt, diesen Broker und das Gateway. Kapitel 3 sagt, was diese Macht begrenzt. Für den Broker ist die Absicherung die eine Aussage, die hier getroffen wird: Sein Credential wird wie das des Eigentümers verwahrt, nur seine eigene Workload Identity kann es abrufen, und jede seiner Aktionen geht in das Audit-Log (Kapitel 14).

Die folgende Sequenz zeigt eine Instanz von `billing-api`, die einen Login anfordert. Achten Sie auf die beiden Stellen, an denen die Anforderung abgelehnt wird: bevor irgendetwas existiert und wenn das Audit-Log nicht erreichbar ist.



In Pseudocode hält die Reihenfolge der Operationen die Sequenz korrekt. Funktionsnamen wie `createLogin` sind beispielhaft, wie an anderen Stellen des Buches.

```
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 }
```

Das Lease wird geschrieben, bevor der Login existiert, und der Audit-Eintrag wird geschrieben, bevor der Login angelegt wird. Fällt die Datenbank aus, nachdem das Lease erfasst wurde, findet der Sweeper ein Lease ohne Login und behandelt es als bereits widerrufen. Ist das Audit-Log nicht verfügbar, wird überhaupt kein Credential ausgestellt. Das ist eine bewusste Entscheidung, und sie bedeutet, dass ein nicht erreichbares Audit-Log neue Instanzen am Start hindert, ein Preis, den Kapitel 14 von der anderen Seite aus behandelt.

Ablauf und Widerruf

Ein Ablauf verhindert nur, dass sich neue Verbindungen authentifizieren. Eine bereits offene Session läuft weiter, nachdem das Passwort abgelaufen ist; der Ablauf hat also zwei Teile: Der Login wird ungültig gemacht, und die offenen Sessions darauf werden geschlossen. Das ist auch der Unterschied zwischen Rotation und **Widerruf**, der dafür sorgt, dass ein Credential oder ein Grant jetzt nicht mehr funktioniert. Rotation ersetzt; Widerruf beendet.

```
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")
```

Die Verlängerung folgt derselben Disziplin. Eine laufende Instanz bittet den Broker, ihr Lease zu verlängern, bevor es endet, und der Broker stimmt nur zu, solange das Lease aktiv ist und die Gesamtlaufzeit unter einer Obergrenze liegt. Ohne Obergrenze macht die Verlängerung aus einem kurzlebigen Credential ein langlebiges, das zufällig neu ausgestellt wird. Die Lebensdauer selbst ist eine Ermessensfrage. Ich kenne die richtige Zahl für Ihr System nicht, und sie hängt von drei Dingen ab: wie lange eine Instanz zwischen Neustarts läuft, wie viele Verbindungen jede Instanz hält und wie lange eine Verlängerung dauert, wenn der Broker unter Last steht. Nur als Beispiel: Denken Sie an ein Lease von einer Stunde, das alle dreißig Minuten verlängert wird.

Der Gewinn gegenüber der A/B-Rotation ist, dass ein durchgesickertes Credential von selbst abläuft. Der Login benennt die Instanz, da er für diese Instanz angelegt wurde, sodass das Datenbank-Log die Frage „Wer hat diese Anweisung ausgeführt?“ bis auf den Workload genau beantwortet. Das erfüllt, was Kapitel 3 versprochen hat: ein Login, der einer einzelnen Workload-Instanz zugeordnet ist.

Workload Identity und IAM-basierte Authentifizierung bei der Datenbank

Es gibt eine Möglichkeit, das gespeicherte Passwort ganz zu entfernen. Manche Plattformen lassen die Datenbank dem Identitätssystem der Plattform selbst vertrauen: Der Workload fordert von der Plattform ein kurzlebiges Token an, und die Datenbank akzeptiert dieses Token anstelle eines Passworts. Cloud-Anbieter bieten das unter Namen wie IAM-Authentifizierung für Datenbanken an, und manche Datenbanken können Tokens eines externen IdP prüfen.

Der Reiz liegt darin, dass es kein Secret gibt, das gespeichert, abgerufen oder rotiert werden müsste. Das Token wird bei Bedarf für den Workload ausgestellt, ist kurzlebig (typischerweise Minuten) und an die Identität gebunden, die die Plattform bereits kennt. Der Widerruf ist Sache der Plattform: Wird der Identität das Recht entzogen, Tokens anzufordern, entstehen keine neuen Tokens, und bereits ausgestellte Tokens werden möglicherweise bis zu ihrem Ablauf akzeptiert. Der Login in der Datenbank existiert weiterhin, ist weiterhin Mitglied seiner Rolle `app_*` und erscheint weiterhin im Log.

Die Kosten sind konkret:

- **Plattformkopplung.** Das Design funktioniert dort, wo die Plattform es unterstützt, und sonst nirgends, sodass ein Team mit Workloads auf zwei Plattformen das Pattern zweimal baut.
- **Mapping.** Jemand muss festlegen, welche Plattform-Identität welchem Login und welcher Rolle entspricht. Dieses Mapping ist ein Grant, und es verdient dasselbe Review wie jeder andere (Kapitel 14).
- **Offene Verbindungen.** Bei gängigen Engines wird das Token beim Öffnen einer Verbindung geprüft. Wie bei jedem Ablauf überlebt eine offene Verbindung ihr Token (siehe „Ablauf und Widerruf“), sodass weiterhin eine Obergrenze für die Lebensdauer von Verbindungen nötig ist (Kapitel 5).

Welcher der drei Ansätze zu wählen ist, hängt weniger von der Mode ab als davon, was die Plattform bietet und was das Team betreiben kann.

Ansatz	Gespeichertes Secret?	Lebensdauer	Voraussetzungen	Hauptkosten
Statisches Credential in einem Secret-Manager, A/B-Rotation	Ja, ein Paar pro Service	Bis zur nächsten Rotation	Secret-Manager, Rotations-Job	Consumer müssen neu lesen; der Rotations-Job muss weiterlaufen
Dynamische Credentials von einem Broker	Nur das des Brokers selbst	Minuten bis Stunden, pro Instanz	Broker, `ledgerly_access_admin`, Lease-Sweeper	Der Broker ist eine Komponente von hohem Wert, die immer verfügbar sein muss
Tokens der Workload Identity	Keines für die Datenbank	Kurzlebig, typischerweise Minuten	Unterstützung durch Plattform und Datenbank	Plattformkopplung; Tokens werden nur beim Verbindungsaufbau geprüft

Keiner dieser Ansätze beseitigt das Credential-Problem; jeder verschiebt es. Der eigene Login des Brokers ist ein statisches Credential, das im Secret-Manager liegt, von der Workload Identity des Brokers abgerufen und per A/B rotiert wird. Das Token-Design verlässt sich darauf, dass das Identitätssystem der Plattform korrekt ist. Was sich ändert, ist, wie viele Secrets es gibt, wie lange jedes nützlich ist und wer sie lesen kann.

TLS und Zertifikatsprüfung

Alles bisher Gesagte betrifft, wo ein Credential aufbewahrt wird und wie lange es gilt. Bevor es die Datenbank erreicht, durchquert es ein Netzwerk, und eine nicht verschlüsselte Verbindung trägt das Passwort und die Daten in einer Form, die alles auf dem Weg lesen kann. Verschlüsselung bei der Übertragung schließt das, und sie ist die Grundvoraussetzung. Sie muss auf jedem Teilstück gelten. Eine Anfrage läuft oft über einen Pooler oder ein Gateway, bevor sie die Datenbank erreicht, und jedes davon ist eine eigene Verbindung mit eigener Verschlüsselung und eigener Zertifikatsprüfung. Das Teilstück vom Client zum Pooler zu verschlüsseln sagt nichts über das Teilstück vom Pooler zur Datenbank.

Es gibt eine zweite Aufgabe, die sich leichter auslassen lässt: zu wissen, dass der Server der richtige ist. Eine Verbindung kann zu einem Ziel verschlüsselt sein, das nicht die Datenbank ist. **Zertifikatsprüfung** bedeutet, dass der Client prüft, ob das Zertifikat des Servers von einer Zertifizierungsstelle ausgestellt wurde, der er vertraut, und ob das Zertifikat den Host nennt, den er erreichen wollte. Ohne sie schützt die Verschlüsselung davor, dass jemand den Verkehr mitliest, und tut nichts dagegen, dass jemand die Verbindung beantwortet.

Entscheidend ist der Verifizierungsmodus des Clients. Die meisten Datenbank-Clients bieten einen Modus, der verschlüsselt, aber nicht verifiziert, und er ist beliebt, weil er mit jedem Zertifikat funktioniert, auch mit abgelaufenen und selbstsignierten. Ich halte diesen Modus für die halbe Arbeit: Er verbirgt den Verkehr vor einem passiven Beobachter und erzeugt das falsche Gefühl, der Server sei der richtige. Abhilfe ist, die Zertifizierungsstelle an jeden Service zu verteilen und den Namen zu prüfen.

Das hat eigene Kosten. Zertifikate laufen ab, und ein abgelaufenes Server-Zertifikat ist ein Ausfall mit exaktem Datum, sodass der Ablauf eine Überwachung und ein geübtes Erneuerungsverfahren braucht. Das Zertifikat der Zertifizierungsstelle an jeden Consumer zu verteilen ist Arbeit, und es muss aktualisiert werden, wenn sich die Zertifizierungsstelle ändert. Client-Zertifikate, bei denen das Zertifikat das Credential ist, funktionieren mit kurzen Lebensdauern nur, wenn etwas sie automatisch ausstellt.

Wenn ein Credential durchsickert

Selbst mit allen oben beschriebenen Kontrollen kann ein Credential durchsickern, und das Design sollte vorab festlegen, was dann geschieht. Der erste Schritt ist, die beiden Fragen zu trennen, die die Panik vermischt: Wie hindern wir das Credential daran, zu funktionieren, und was hat es getan, solange es funktionierte?

Bei einem dynamischen Credential ist die erste Antwort ein Widerruf: das Lease beenden, die Sessions schließen, und der Login ist innerhalb der Zeit weg, die der Broker zum Handeln braucht. Bei einem statischen Credential in einem A/B-Setup ist ein vermutetes Leck nicht die planmäßige Rotation. Der planmäßige Job wartet, bis der alte Login abgeflossen ist, und ein durchgesickerter Wert bleibt so lange gültig, wie irgendjemand

eine Session darauf hält. Bei einem Leck kappen Sie den durchgesickerten Login zuerst: seine Rollen entfernen, seine Sessions schließen und sein Passwort ändern, ohne auf den Abfluss zu warten, dann einen neuen Wert für den inaktiven Login veröffentlichen und den aktiven Zeiger umschalten. Instanzen, die noch den alten Login halten, verlieren ihre Verbindungen und schlagen fehl, bis sie das Secret neu lesen. Das sind Verfügbarkeitskosten, und ich würde sie zahlen: Ein durchgesickertes Credential, das weiter funktioniert, ist das größere Problem. Ein Service, der sein Secret nicht neu lesen kann, braucht einen Neustart, und das Team sollte wissen, welche Services das sind, bevor es darauf ankommt. Beim gemeinsamen Admin-Passwort im Ausgangszustand ist die erste Antwort der Ausfall, mit dem dieses Kapitel begonnen hat, und das ist das stärkste Argument dafür, die Arbeit vor einem Incident zu erledigen.

Die zweite Frage beantwortet das Log. Ein Login, der einer Workload-Instanz zugeordnet ist, lässt Sie die Anweisungen sehen, die er ausgeführt hat, die Tabellen, die er berührt hat, und die Stunden, in denen er aktiv war. Das begrenzt die Prüfung auf den Auswirkungsbereich einer Rolle statt auf die ganze Datenbank. Das schrittweise Vorgehen, die Reihenfolge der Maßnahmen und die Übung dafür gehören in Kapitel 15, das den Zugriff bei Incidents behandelt. Hier ist der Punkt, dass ein Team, das den Widerruf als Routineschritt aufgebaut hat, ihn an einem schlechten Tag einsetzen kann, ohne ihn erst erfinden zu müssen.

Regelwerke zum Datenschutz und zu Zahlungskarten fragen tendenziell, wie Zugriff gewährt wird, wie lange er gilt und wie er widerrufen wird. Die hier beschriebenen Leases, Widerrufsschritte und Audit-Einträge sind die Aufzeichnungen, die diese Fragen beantworten.

Ledgerlys Migrationsplan

Der Wechsel vom gemeinsamen Admin-Passwort zu kurzlebigen Credentials ist eine Folge kleiner, umkehrbarer Schritte. Ich würde sie danach ordnen, wie viel Risiko jeder Schritt beseitigt und wie wenig er kaputtmachen kann.

1. **Jede Kopie des Admin-Passworts inventarisieren.** Services, Pipeline-Variablen, der Export-Schritt, der Passwortmanager, gespeicherte Ver-

bindungen in Client-Tools. Die Liste ist der Arbeitsplan. Das Inventar von Kapitel 1 ist der Ausgangspunkt.

2. **Pro Service einen Login in dessen Rolle `app_*` anlegen.** Verwenden Sie die Rollen aus Kapitel 3. In dieser Phase sind die Credentials statisch, aber jedes ist getrennt, im Secret-Manager gespeichert und wird über die Identität gelesen.
3. **Services einzeln vom Admin-Login lösen.** Beginnen Sie mit `reporting-job`, dessen Auswirkungsbereich am kleinsten ist, dann `invoice-worker`, `admin-console` und `billing-api`. Stellen Sie `payments-service` zuletzt um, weil er der einzige Service ist, der `payment_tokens` und `bank_accounts` erreicht, und ein Fehler dort am meisten kostet. Jeder Wechsel ist ein Deployment, das sich per Rollback zurücknehmen lässt.
4. **Migrationen und dem Export-Schritt eigene Logins geben.** Die Pipeline verwendet einen kurzlebigen Login in `ledgerly_migrator` (Kapitel 11); die Rolle selbst hält dauerhafte Eigentümerprivilegien, und nur das Credential ist kurzlebig. Der Export-Schritt verwendet `app_vendor_export` (Kapitel 13). Keiner von beiden nutzt nach diesem Schritt noch den Admin-Login.
5. **Jeden Service auf A/B-Rotation umstellen.** Lassen Sie jeden Service sein Secret nach einem Authentifizierungsfehler neu lesen, führen Sie die erste Rotation in einer ruhigen Phase durch und beobachten Sie die Abflussprüfung. Ein Service, bei dem der Abfluss ausbleibt, ist ein Befund, und dieser Befund ist der Sinn dieses Schritts.
6. **Den Broker für die Services einführen, die sich leicht verlängern lassen.** Beginnen Sie mit einem Service, der oft neu startet und wenige Verbindungen hält. Fügen Sie die übrigen hinzu, während das Team lernt, wie sich die Leases verhalten. Behalten Sie die statische A/B-Rotation als Fallback.
7. **Den gemeinsamen Admin-Login stilllegen.** Entfernen Sie ihn aus jeder Konfiguration und aus dem Passwortmanager. Sobald das Eigentum auf `ledgerly_owner` übergegangen ist (Kapitel 3), entziehen Sie ihm die Privilegien und löschen oder deaktivieren Sie ihn. Erweist sich der Admin als der Superuser der Engine, wie es bei dem Master-Nutzer eines verwalteten Services oft der Fall ist, wird er zu dem, was Kapitel 3 das versiegelte Instrument nennt. Das ist ein Credential, das niemand in der täglichen Arbeit nutzt. Es wird dort aufbewahrt, wo

das Platform-Team es abrufen kann, und jeder Abruf wird aufgezeichnet. Ändern Sie sein Credential einmal, nachdem das gesamte Inventar bereinigt ist.

Ein Team könnte nach Schritt fünf aufhören und stünde trotzdem deutlich besser da: Jeder Service hat sein eigenes Credential, liest es über eine Identität und rotiert es ohne Ausfall. Der Broker ist eine Ergänzung für Teams, die ihn betreiben können.

Abwägungen und Kosten

Jeder Schritt in diesem Kapitel verlagert Kosten von einem Incident in die Routine. Hier wird die Routine teuer.

Abhängigkeit vom Secret-Manager. Sobald Credentials aus einem Secret-Manager kommen, hindert ein nicht erreichbarer Secret-Manager neue Instanzen am Start und, bei kurzen Leases, schließlich auch laufende Instanzen an der Verlängerung. Damit ist er Teil der Verfügbarkeit des Datenbankpfads. Betreiben Sie ihn mit der Sorgfalt, die Sie der Datenbank widmen: mehr als eine Instanz, eine getestete Wiederherstellung und eine Rufbereitschaftsrotation, die von seiner Existenz weiß. Ein Credential für die Dauer eines Leases zwischenzuspeichern mildert einen kurzen Ausfall und verlängert die Zeit, in der ein widerrufenes Credential noch benutzt werden kann.

Connection-Storms. Verlängert jede Instanz eines Services in derselben Minute oder startet sie nach einem Deployment gemeinsam neu, sieht die Datenbank einen Schwall neuer Verbindungen und Authentifizierungen. Authentifizierung ist nicht umsonst, und ein selbstverursachter Schwall kann die Datenbank für jede andere Rolle verlangsamen, ein Connection-Storm ohne äußere Ursache. Staffeln Sie die Verlängerungen mit Jitter, halten Sie den Connection-Pool zwischen den Verlängerungen stabil, und lassen Sie Kapitel 5 bestimmen, wie viele Verbindungen eine Rolle halten darf.

Schwierigere Fehlersuche. Ein Login, der eine Stunde lang existiert, ist weg, wenn jemand am nächsten Morgen das Log liest, und ein Name wie `billing-api-4f2a` bedeutet ohne den Lease-Datensatz nichts. Bewahren Sie Leases und ihre Identität so lange auf, wie das Audit-Log aufbewahrt

wird, und geben Sie der diensthabenden Person eine Möglichkeit, vom Login zur Instanz nachzuschlagen. Fehlt diese Möglichkeit, macht das kurzlebige Credential die Untersuchung langsamer, nicht schneller.

Ein Broker von hohem Wert. Der Login des Brokers kann Logins in jeder Anwendungsrolle anlegen, sodass sich der Aufwand für einen Angreifer lohnt. Halten Sie seinen Code klein und geprüft, und widmen Sie ihm die Aufmerksamkeit, die Sie dem widmen, was er schützt; ein Broker, der weniger bekommt, ist selbst ein Risiko.

Wann man das nicht einsetzen sollte. Ein Team mit einem einzigen Service, einer Handvoll Entwicklern und einem verwalteten Speicher für Secrets kann mit einem eigenen statischen Credential gut bedient sein, das im Secret-Manager liegt und zweimal im Jahr von Hand per A/B rotiert wird. Ein Broker ist ein weiterer Service, der betrieben werden muss, und ein weiteres System, das das Team lernen muss; unterhalb einer gewissen Größe verursachen die zusätzlichen beweglichen Teile mehr Ausfälle, als sie verhindern. Wenn Sie nicht benennen können, wer alarmiert würde, wenn der Broker ausfällt, würde ich warten. Tokens der Workload Identity sind der günstigste Schritt, wenn die Plattform sie anbietet, und brauchen trotzdem ein geprüftes Mapping.

Häufige Fehler

- **Den Secret-Manager für die ganze Antwort halten.** Es ist verlockend, weil der Schritt sichtbar und schnell ist. Das gemeinsame Passwort bleibt gemeinsam, wenn jeder Service denselben Pfad liest. Geben Sie jedem Service seinen eigenen Pfad und seinen eigenen Login, und prüfen Sie, wer jeden von ihnen lesen darf.
- **Eine Rotation, die niemand konsumiert.** Ein Job rotiert den Wert, aber Services lesen ihn beim Start und nie wieder, sodass der neue Wert ungelesen daliegt, bis der alte stillgelegt wird und etwas ausfällt. Testen Sie, dass Consumer eine Rotation übernehmen, und machen Sie die Abflussprüfung zum Teil des Jobs.
- **Auf einen eingebauten Wert zurückgreifen.** Eine „zuletzt funktionierende“ Kopie im Image macht den Start zuverlässig und bringt die Datei zurück, die das Design beseitigt hat. Lehnen Sie den Start ab, lösen Sie einen Alarm aus und reparieren Sie die Abhängigkeit.

- **Verschlüsseln ohne zu verifizieren.** TLS einzuschalten erfüllt eine Checkliste, und der reine Verschlüsselungsmodus lässt die Verbindung funktionieren. Verifizieren Sie das Zertifikat und den Namen des Hosts, und überwachen Sie das Ablaufdatum.
- **Lebensdauern ohne Obergrenze.** Eine Verlängerung ohne Limit baut ein langlebiges Credential wieder auf. Begrenzen Sie die Gesamtlebensdauer eines Leases, und lassen Sie die Instanz sich mit ihrer Identität neu authentifizieren.
- **Die Rolle des Brokers für klein halten.** `ledgerly_access_admin` ist weiter gefasst, als die Aufgabe des Brokers es verlangt, sodass ein übernommener Broker so weit reichen kann wie die Rollen, die er verwaltet. Beschränken Sie, welche Principals ihn aufrufen dürfen und welche Rollen `app_*` jeder anfordern darf, und prüfen Sie seine Audit-Einträge. Betrachten Sie die weit gefasste Rolle als Kosten, die Sie akzeptiert haben, nicht als Begrenzung.

Zentrale Erkenntnisse

- Jede Kopie eines Credentials ist ein Leser; die nützlichste Änderung ist daher, die Zahl der Kopien zu verringern und jedem verbleibenden Lesevorgang einen Namen zu geben.
- Rotation ist schwierig, weil ein Credential geteilter Zustand ist; zwei Logins pro Service machen es möglich, einen zu ändern, während der andere weiterarbeitet.
- Eine Rotation, die die Consumer nicht übernehmen oder die niemand ausführt, ist ein Kalendereintrag und keine Kontrolle; deshalb gehören die Abflussprüfung und ein Alarm für einen Rotations-Job, der nicht mehr läuft, zum Design.
- Dynamische Credentials geben jeder Workload-Instanz einen eigenen Login mit eingebautem Ende, zum Preis eines Brokers, eines Lease-Datensatzes und einer Abhängigkeit vom Secret-Manager.
- Die Workload Identity beseitigt das gespeicherte Passwort, wo die Plattform es unterstützt, und verlagert das Vertrauen auf die Plattform und das Mapping zwischen Identität und Login.
- Das Verifizieren von Zertifikat und Name des Servers macht aus TLS einen Schutz für das Credential; Verschlüsselung ohne Verifizierung

verbirgt den Verkehr und lässt die Identität des Servers ungeprüft.

- Wenn ein Credential durchsickert, entscheiden die Geschwindigkeit des Widerrufs und die Klarheit des Logs darüber, wie eingegrenzt das Ereignis bleibt, und das hängt von Arbeit ab, die vor dem Leck erledigt wurde.

Credentials entscheiden, wer hineinkommt; sie sagen nichts darüber, wie viele Verbindungen sie öffnen oder wie lange eine Anweisung den Primary belegen darf. Kapitel 5 macht die Verbindung selbst zur Richtlinie: Limits pro Rolle, Pooling, Timeouts und die Labels, die eine außer Kontrolle geratene Abfrage leicht auffindbar machen.

Kapitel 5 — Verbindungen als Richtlinie

Stellen Sie sich Ledgerly in einer ruhigen Dienstagnacht vor. Ein Teammitglied hat die Parallelität der nächtlichen Aggregation in `reporting-job` erhöht, damit sie vor dem Morgen fertig wird, eine Änderung, die im Review harmlos aussah. Um 01:40 Uhr startet der Job, öffnet für jeden seiner Worker-Threads eine Verbindung und beginnt, mit dem gemeinsamen Login `admin` `payments` und `invoices` vom Primary zu lesen. Innerhalb weniger Minuten gibt `billing-api` Fehler an Kunden zurück, in einer Zeitzone, in der es bereits Morgen ist. Robin, in Rufbereitschaft, wird wegen eines Ausfalls mit Kundenwirkung alarmiert.

Robin öffnet die Datenbank und stellt fest, dass nichts kaputt ist. Der Primary läuft, die Plattenaktivität ist normal, und keine Anweisung hat etwas falsch gemacht. Der Server hat eine feste Zahl von Connection-Slots, der Report hat die meisten davon belegt, und die API wartet an der Tür. Weil alle Aufrufer denselben Login verwenden, kann die Datenbank nicht sagen, dass der Report die Ursache ist. Sie kann nur sagen, dass die Slots voll sind.

Dieses Kapitel beantwortet eine Frage, die zwischen Identität und Daten liegt: Wie machen wir die Verbindung selbst zu einem Ort, an dem Richtlinien angewendet werden? Im Fünf-Teile-Modell aus Kapitel 1 wirkt es auf zwei Teile. Die Verbindung ist der Pfad, weil der Pooler bestimmt, wie Services ankommen und wie viele von ihnen es dürfen. Sie ist außerdem die erste Grenze der Ausführung, weil Limits und Timeouts festlegen, wie viel ein Aufrufer und eine Anweisung allen anderen wegnehmen dürfen. Der Schritt aus dem laufenden Beispiel lautet: Verbindungen werden durch einen Pooler sowie durch Limits und Timeouts pro Rolle geformt.

Das Problem und warum es schwierig ist

Eine Verbindung ist zuerst eine Ressource. Bei den meisten Engines kostet jede den Server Speicher und einen Anteil am Scheduling, und der Server erlaubt nur eine feste Zahl davon. Das Limit ist eine Eigenschaft des Servers, nicht des Aufrufers, sodass der Erste, der viele anfordert, auch viele bekommt. Nichts im Standard-Setup sagt, dass der nächtliche Report weniger Slots haben sollte als die API mit Kundenwirkung.

Ein Passwort berührt davon nichts. Es entscheidet, wer hineinkommt; es sagt nichts darüber, wie viele Slots jemand hält und wie lange. Deshalb braucht ein Buch, das sich um mehr als die Verteilung von Credentials dreht, ein Kapitel über Verbindungen.

Drei Gewohnheiten verschärfen das Problem. Erstens ist die Zahl der Verbindungen ein Produkt, keine Summe: die Instanzen eines Services mal die Größe des Pools jeder Instanz, und das noch einmal, sobald ein Autoscaler eine Instanz hinzufügt. Angenommen, ein Team stellt für einen Service, der auf zehn Instanzen skaliert, einen Pool von zwanzig ein. Dann hat es zweihundert Verbindungen angefordert und das vermutlich nicht bewusst entschieden. Zweitens wird eine Verbindung so lange gehalten, wie ihre Anweisung läuft, sodass eine langsame Abfrage die ganze Zeit einen Slot belegt, und Slots sind das, was der nächste Aufrufer braucht. Drittens versucht ein scheiternder Aufrufer tendenziell, sich mehr anzustrengen. Ein Client, der in einen Timeout läuft und sofort einen Retry durchführt, macht aus einer langsamen Datenbank eine Menschenmenge an der Tür.

Die naheliegende Reaktion ist, das Server-Limit anzuheben. Das hilft eine Woche lang und macht dann den nächsten Incident größer. Mehr Verbindungen bedeuten mehr gleichzeitige Arbeit auf demselben Primary, und die Datenbank wird nicht schneller darin, hundert Anweisungen gleichzeitig auszuführen, nur weil hundert erlaubt sind.

Die Kontrollen liegen an mehreren Stellen, die niemand gemeinsam verantwortet: in der Rollendefinition, im Pool der Anwendung, in einem möglichen Pooler und in den Einstellungen, die bestimmen, wie lange eine Anweisung laufen darf. Verfügbarkeit ist eine der Eigenschaften, die im Bedrohungsmodell von Kapitel 2 auf dem Spiel stehen, und Unfälle dominieren sie. Ein gemeinsamer Login macht jeden Unfall zum Unfall aller.

Im Ausgangszustand ziehen also alle fünf Services und jeder Engineer im VPN aus einem Pool von Server-Slots unter einem Namen. Es gibt keine Timeouts außer dem, was die jeweilige Client-Bibliothek als Default mitbringt, und keine Aufzeichnung, welcher Aufrufer welche Verbindung hält. Die folgenden Abschnitte ersetzen das durch ein Budget pro Rolle, einen Pooler, der es durchsetzt, Timeouts, die eine Obergrenze dafür setzen, wie lange irgendetwas einen Slot halten darf, und ein Label an jeder Verbindung, das sagt, wem sie gehört.

Ein Verbindungsbudget pro Rolle

Der erste Schritt ist der günstigste. Kapitel 3 hat jedem Service seine eigene Anwendungsrolle gegeben, was bedeutet, dass ein Limit endlich an einen Namen geknüpft werden kann. Ein Limit pro Rolle sagt: Dieser Principal darf höchstens so viele Verbindungen halten, gleichgültig, wie viele Instanzen er betreibt oder wie stark er anfordert. Erreicht er das Limit, werden neue Verbindungen dieser Rolle abgelehnt, und jede andere Rolle arbeitet weiter.

Diese Ablehnung ist die Richtlinie. Sie verschiebt den Fehler von „die Datenbank ist für alle voll“ zu „diesem Aufrufer sind die Slots ausgegangen“, und das Zweite ist eine sehr viel bessere Alarmierung für Robin. Der Auswirkungsbereich eines gierigen Aufrufers wird zum Aufrufer selbst, und die Ablehnung nennt den Aufrufer, was die Hälfte der Zurechenbarkeit desselben Schritts ist.

Die Zahlen brauchen eine Begründung, keine Formel. Ich würde vom Limit des Servers abwärts arbeiten. Zuerst ziehen Sie eine Reserve ab, die sonst nichts nutzen darf, denn auch die Menschen und Werkzeuge, die das System reparieren, müssen sich verbinden können. Die Reserve deckt Plattform-Sessions und die Logins der drei Principals ab, die `ledgerly_access_admin` nutzen (der Pipeline-Schritt, der Rollendefinitionen anwendet, der Credential-Broker und das Access-Gateway). Dann verteilen Sie den Rest auf die Anwendungsrollen, danach, was jede in ihrer Spitzenzeit braucht, plus etwas Spielraum. Die folgende Tabelle ist eine Illustration für den Primary. Die Zahlen sind hypothetisch, keine Empfehlung.

Principal	Rolle	Verbindungslimit (beispielhaft)	Warum diese Größe
`billing-api`	`app_billing_api`	80	Meistgenutzter Service; bedient die Web-App und die öffentliche API
`invoice-worker`	`app_invoice_worker`	30	Queue-gesteuert; die Nebenläufigkeit ergibt sich aus der Zahl der Worker
`payments-service`	`app_payments_service`	30	Gleichmäßig; spricht zwischen Anweisungen mit externen Providern
`admin-console`	`app_admin_console`	15	Enge, auditierte Aktionen weniger Menschen
Migrationen	`ledgerly_migrator`	3	Eine Migration zur Zeit, aus der Pipeline
Menschliche Sessions auf dem Primary	Session-Logins der Art JIT und Break-Glass, vom Gateway angelegt	8	Hier laufen nur Sessions mit einem genehmigten Antrag (Kapitel 6); Routine-Lesezugriffe gehen an `ledgerly_replica_2`. Die Rollen `human_jit_write` und `human_breakglass` tragen kein Limit, weil kein Login Mitglied der ersten ist und die zweite pro Session gewährt wird
`reporting-job`	`app_reporting_job`	10	Nur im Übergang: Er liest im Ausgangszustand und während des Umzugs vom Primary, und die Zeile geht im Zielzustand auf null
Reserve	Zugriffsverwaltung und Plattform	10	Nur für Reparatur und für die Ausstellung von Logins
Nicht zugewiesen	keine	14	Spielraum, damit ein Rundungsfehler kein Ausfall wird

Angenommen, der Server erlaubt 200 Verbindungen. Die zugewiesenen Zeilen summieren sich auf 186, und die verbleibenden 14 bleiben bewusst unzugewiesen. Die routinemäßigen menschlichen Sessions stehen nicht in dieser Tabelle, weil sie nicht auf dem Primary laufen. Ihr Budget wird auf `ledgerly_replica_2` festgelegt, wohin die Routine-Lesezugriffe gehen;

sagen wir, 20 Slots dort. Die Zeile `reporting-job` existiert nur, solange der Report noch den Primary liest. Im Zielzustand liest er von `ledgerly_replica_2`, mit einem eigenen Budget auf diesem Host, das auf der Verbindungsebene im Pooler gesetzt wird (Kapitel 6 behandelt den Umzug). Die Rollendefinitionen könnten die Limits in einer Notation wie dieser tragen, die beispielhaft ist und zu keiner Engine gehört:

```
// 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;
```

> **Hinweis:** Engines unterscheiden sich darin, woran ein Limit oder eine Einstellung hängt. Manche zählen Verbindungen pro Rolle, manche pro Login und manche nur pro Datenbank oder pro Server. Bei manchen Engines gilt ein Limit oder ein Default, das an einer Rolle geschrieben ist, nur für Verbindungen, die sich als die Rolle selbst anmelden, und wird nicht an die Logins vererbt, die ihre Mitglieder sind. Kapitel 4 gibt jeder Service-Instanz einen eigenen dynamischen Login, ein Mitglied der Rolle `app_*`; bei diesen Engines würden das Limit auf Rollenebene und die Timeouts auf Rollenebene weiter unten also stillschweigend nicht greifen.

Stattdessen können zwei Stellen sie tragen: der Broker, der beim Anlegen jedes Logins ein Limit pro Login und dieselben Einstellungen setzen kann, und der Pooler, der die Einstellungen tragen kann, aber möglicherweise die Logins einer Rolle nicht zu einer Gesamtzahl summieren kann. Der Abschnitt zum Pooler unten zeigt drei Wege, eine Gesamtzahl pro Rolle zu erhalten, wenn er es nicht kann.

Zwei Eigenschaften machen die Tabelle nützlich. Die Limits werden dort geprüft, wo die Verbindung beginnt, sodass ein Aufrufer die Ablehnung sofort erfährt und nicht erst nach der ersten langsamen Anweisung. Und die Reserve liegt außerhalb jeder Anwendungsrolle, sodass das Platform-Team sich in der schlimmsten Nacht noch anmelden kann, um Dinge zu reparie-

ren. Ein Budget, das den ganzen Server füllt, hat denselben Fehler wie ein gemeinsames Admin-Passwort: Es funktioniert bis zu dem Moment, in dem es jemand am dringendsten braucht.

Die Summen müssen nicht konservativ sein. Wenn `billing-api` und `payments-service` selten gleichzeitig ihre Spitze erreichen, können Sie die Limits der Anwendungen zu mehr summieren lassen, als der Server erlaubt, und hinnehmen, dass zwei gleichzeitige Spitzen kollidieren könnten. Ich würde das nur tun, wenn die Reserve zurückgehalten wird und erst nachdem das Team die tatsächliche Nutzung eine Zeit lang beobachtet hat. Der Default ist ein Budget, das aufgeht, weil ein Budget, das sich nicht verletzen lässt, um drei Uhr morgens leichter zu durchdenken ist.

Pooling: wo die Verbindungen geteilt werden

Ein Pool hält eine Menge offener Verbindungen und verleiht sie an Aufrufer, sodass nicht jede Anfrage die Kosten einer neuen Verbindung trägt und die Datenbank eine begrenzte Zahl davon sieht. Es gibt zwei Stellen, an denen man einen platzieren kann.

Fast jeder Service hat bereits einen Pool in seiner Datenbankbibliothek, und er sollte nach der Arbeit bemessen werden, die eine Instanz gleichzeitig leisten kann, nicht nach dem Traffic, den sie zu bedienen hofft. Ein kleiner Pool, der Aufrufer kurz warten lässt, ist für die Datenbank freundlicher als ein großer, und er sollte nach kurzer Wartezeit mit einem Fehler aufgeben, den der Service in eine klare Antwort übersetzen kann. Beginnen Sie klein, beobachten Sie, wie lange Aufrufer warten, und erhöhen Sie die Größe anhand von Belegen.

Der Pooler vor der Datenbank

Ein **Pooler** ist ein Datenbank-Proxy, dessen Hauptaufgabe darin besteht, viele Client-Verbindungen anzunehmen und sie über eine kleinere Zahl von Server-Verbindungen zu multiplexen. PgBouncer und ProxySQL sind gängige Beispiele. Der Service verbindet sich mit dem Pooler, als wäre er die Datenbank, und der Pooler entscheidet, wann eine echte Verbindung verliehen wird.

Die Frage, die das Buch immer wieder stellt, gilt auch hier: Welchen Namen sieht die Datenbank? Kapitel 4 hat jeder Service-Instanz ihren eigenen dynamischen Login gegeben, vom Credential-Broker angelegt und Mitglied der Rolle `app_*` des Services. Der Pooler muss zu diesem Modell passen, und ich würde ihn in dieser Form bauen. Die Instanz legt ihren eigenen Login und ihr eigenes Credential beim Pooler vor. Der Pooler prüft beides gegen die Datenbank und hält Server-Verbindungen unter demselben Login. Er hat keinen eigenen Passwortspeicher und keinen eigenen Login, den alle Clients teilen.

Ein rotiertes oder abgelaufenes Credential erreicht den Pooler über die Datenbank, nicht über den Pooler. Endet ein Lease oder wird es widerrufen, löscht der Broker den Login (Kapitel 4), sodass der nächste Versuch des Poolers, sich zu authentifizieren oder unter diesem Login eine Server-Verbindung zu öffnen, abgelehnt wird, und er schließt die im Leerlauf gehaltenen Server-Verbindungen dafür.

Wie ein Limit pro Rolle trotzdem gilt, dafür gibt es keine allgemeine Antwort, und ich würde nicht davon ausgehen, dass Ihr Pooler eine liefern kann. Viele Pooler führen ihre Limits pro Paar aus Login und Datenbank, und manche fügen ein Limit pro Login oder pro Datenbank hinzu. Mir ist keine allgemeine Fähigkeit bekannt, die die Verbindungen vieler verschiedener Logins zu einer Obergrenze auf Rollenebene summiert; lesen Sie also die Dokumentation Ihres Poolers, statt diesem Kapitel zu glauben. Kann Ihrer es nicht, muss die Gesamtzahl pro Rolle von anderswo kommen. Es gibt drei Stellen, und jede hat ihren Preis.

- **Der Pooler verbindet sich upstream mit einem stabilen Login pro Rolle.** Das dynamische Credential der Instanz authentifiziert den Client dann nur gegenüber dem Pooler, und das Limit des Poolers pro Login ist genau das Limit der Rolle. Der Preis ist das Identitätsmodell aus Kapitel 3:
  - Das Datenbank-Log zeigt einen Namen pro Rolle, sodass die Zuordnung zu einer Instanz ins Log des Poolers und in das weiter unten beschriebene Anwendungs-Label wandert.
  - Der Pooler muss dynamische Credentials selbst prüfen, was ihn zu einem Consumer des Lease-Zustands des Brokers macht.
  - Er hält außerdem ein langlebiges Credential pro Rolle, das zu schützen und wie in Kapitel 4 beschrieben per A/B zu rotieren ist.

    - Den Login einer Instanz beim Broker zu widerrufen schließt deren Server-Verbindungen nicht mehr; der Pooler muss die Sessions dieses Clients beenden.

- **Die Datenbank setzt ein Limit pro Rolle durch, wo die Engine es für Mitglieds-Logins unterstützt.** Es ist exakt, es sitzt dort, wo nichts es umgehen kann, und die Instanz-Logins samt Zuordnung bleiben erhalten. Der Preis ist, dass die Unterstützung variiert, wie der Hinweis oben sagt. Wo es funktioniert, kommt die Ablehnung von der Datenbank, nachdem die Verbindung bereits eine Authentifizierung gekostet hat, und nichts wird gemultiplext.
- **Der Broker begrenzt, wie viele Logins er pro Rolle ausstellt.** Mit einem Limit pro Login, das beim Anlegen jedes Logins gesetzt wird, kann die Gesamtzahl der Rolle die Zahl der Logins mal dieses Limit nicht überschreiten. Der Preis ist, dass dies eine Obergrenze ist und keine exakte Zählung, und dass die Begrenzung zur Instanzobergrenze des Autoscalers passen muss. Ein Scale-out über die Begrenzung hinaus wird zu einem Deployment, das keine Credentials bekommt, was ein Verfügbarkeitsproblem ist, und Verlängerungen, bei denen sich ein alter und ein neuer Login kurz überlappen, brauchen Platz unter der Begrenzung.

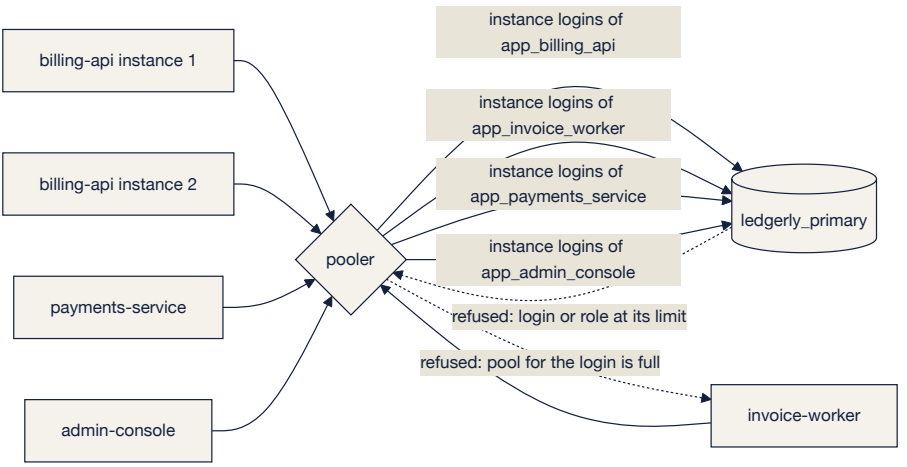
Ich würde zuerst das Limit auf Datenbankseite versuchen, wo die Engine es für Mitglieds-Logins berücksichtigt. Wo sie es nicht tut, bewahrt die Begrenzung durch den Broker das Identitätsmodell und kostet eine Obergrenze, die gepflegt werden muss. Den einzelnen Upstream-Login pro Rolle würde ich nur wählen, wenn die Verbindungszahlen ein Multiplexen über Instanzen hinweg erzwingen, und ich würde aufschreiben, was das Log dann nicht mehr sagt. Alle drei sind besser als das Design, das jeden Client jeder Rolle auf einen gemeinsamen Login abbildet. Es erschafft das gemeinsame Admin-Konto eine Ebene tiefer neu: Das Log zeigt wieder nur einen Namen, und ein Limit gilt für alle auf einmal.

In der empfohlenen durchreichenden Variante bringt der Pooler wenig Multiplexing. Im Transaktionsmodus teilen sich die Clients einer Instanz die Server-Verbindungen dieser Instanz, aber zwei Instanzen teilen nicht miteinander, sodass die Gesamtzahl einer Rolle mindestens der Zahl der aktiven Instanzen mal einem kleinen Minimum entspricht. Was der Pooler bringt, sind drei andere Dinge.

- **Zentrale Ablehnung.** Eine Stelle sagt nein, pro Login und pro Pool, für jede Client-Bibliothek auf dieselbe Weise.
- **Dieselben Limits für jeden Service.** Die Pool-Größen und Wartezeiten in jedem Service sind nicht mehr die einzige Verteidigungslinie.
- **Ein zentraler Netzwerkübergang.** Services gelangen nur über den Pooler in die Datenbank. Die Netzwerkrichtlinie, die festlegt, welche Hosts eine TCP-Verbindung öffnen dürfen und die Kapitel 6 definiert und Kapitel 7 schreibt, lässt die Hosts des Poolers zu und sonst niemanden, sodass ein Credential, das aus einem Service durchsickert, von überall dort nutzlos ist, wo der Pooler nicht erreichbar ist. Die Reichweite begrenzt ein Leck, wie Kapitel 4 sagt, und der Pooler ist die Stelle, an der sich diese Grenze leicht formulieren lässt.

Die Kosten sind ein neues System im Pfad, das in den Abwägungen unten behandelt wird. Für einen einzelnen kleinen Service mit einer Instanz genügen der Pool der Anwendung und ein Limit pro Rolle, und der Pooler lohnt sich erst, wenn die Zahl der Instanzen oder der Bedarf an einem zentralen Übergang etwas anderes verlangt.

Das folgende Diagramm zeigt die durchreichende Variante, bei der die Instanz-Logins für die Datenbank sichtbar bleiben. Achten Sie auf die beiden Stellen, an denen ein Aufrufer abgelehnt wird: am Pooler, wenn der Pool für einen Login voll ist, und an der Datenbank, wenn ein Login oder, wo die Engine es unterstützt, die Rolle an ihrem Limit ist.



Session-Modus und Transaktionsmodus

Ein Pooler verleiht eine Server-Verbindung für einen Zeitraum, und die Länge dieses Zeitraums ist sein **Modus**. Im Session-Modus behält der Client dieselbe Server-Verbindung vom Verbinden bis zum Trennen; der Pooler spart nichts außer den Kosten des Verbindens. Im Transaktionsmodus hält der Client eine Server-Verbindung nur für die Dauer einer Transaktion, und die nächste Transaktion kann auf einer anderen laufen. Der Transaktionsmodus multiplext weit mehr Clients über wenige Verbindungen, und deshalb ist er attraktiv.

Er hat einen Preis, der leicht übersehen wird, weil er die Autorisierung betrifft und nicht die Performance. Alles, was ein Client in seiner Session setzt, lebt auf einer Server-Verbindung, und im Transaktionsmodus besitzt der Client keine. Einstellungen, die außerhalb einer Transaktion vorgenommen werden, können bei der nächsten Anweisung verschwinden oder, schlimmer, auf einer Verbindung bleiben, die der Pooler an jemand anderen verleiht. Zu den Dingen, die vom Session-Zustand abhängen, gehören temporäre Tabellen, Prepared Statements bei vielen Setups, Advisory Locks, Timeouts auf Session-Ebene und jeder Kontextwert, den Ihr eigenes Design in der Session ablegt.

Dieser letzte Punkt berührt das Buch unmittelbar. In Kapitel 3 kann eine Zeilenregel von einem Wert wie `current_case_customer()` abhängen, und `app_admin_console` trägt die Identität eines menschlichen Principals in das Audit-Log und in `audit_events`. Wird einer der beiden Werte einmal pro Session gesetzt und später gelesen, kann er im Transaktionsmodus fehlen oder, im schlimmen Fall, zum vorherigen Client gehören.

> **Warnung:** Im Transaktionsmodus kann ein Kontextwert, der auf Session-Ebene gesetzt wurde, auf einer Server-Verbindung zurückbleiben und von der Transaktion des nächsten Clients gelesen werden. Eine Zeilenregel oder ein Audit-Eintrag, der ihn liest, handelt dann womöglich für den falschen Kunden oder die falsche Person. Setzen Sie solche Werte innerhalb der Transaktion und auf sie beschränkt, und sorgen Sie dafür, dass der Code, der sie liest, fehlschlägt, wenn der Wert fehlt.

Das Pattern, das trägt, besteht darin, den Kontext innerhalb der Transaktion zu setzen und ihn mit der Transaktion enden zu lassen. Der nächste Block zeigt die Form, und sie ist so geschrieben, dass sie korrekt bleibt, wenn die Arbeit zur Hälfte scheitert.

```
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
```

Das Datenbank-Log zeigt weiterhin den Login der Instanz, aber im Transaktionsmodus teilen sich die Clients einer Instanz deren Server-Verbindungen, sodass das Log nicht sagen kann, welcher Thread oder welche Anfrage der Instanz einen Slot gehalten hat. Um das herauszufinden, brauchen Sie die eigene Sicht des Poolers auf seine Clients, und sie muss vor dem ersten Incident geplant sein. Deshalb sind die nächsten Abschnitte zu Timeouts und Benennung wichtig.

Menschen nutzen diesen Pooler nicht. Sie erreichen die Datenbank über das Access-Gateway, das für jede Session einen Login anlegt (Kapitel 7). Ihre Limits sind das Thema des letzten Teils des nächsten Abschnitts, denn ein Mensch kann etwas, das ein Service nicht kann: das Limit aufheben.

Timeouts als Richtlinie

Ein Slot wird so lange verliehen, wie eine Anweisung oder eine Transaktion läuft, und bei den meisten Engines beendet standardmäßig nichts davon eine von beiden. Timeouts sind der Weg, auf dem das Platform-Team festlegt, wie lange eine Rolle eine Ressource halten darf, und sie sind der Teil dieses Kapitels, der die Ausführung begrenzt: Eine Anweisung, die zu lange läuft, wird dort abgebrochen, wo sie läuft, und der Slot geht zurück.

Sie gehören in die Rollendefinition und die Pool-Konfiguration und nicht in das Wohlwollen jeder Client-Bibliothek, weshalb dieses Kapitel sie als Richtlinie behandelt.

Drei Timeouts gehören der Datenbank und zwei dem Pool.

Timeout	Gehört zu	Was er stoppt	Wer ihn spürt
Statement	Datenbank	Eine Abfrage, die weit länger läuft, als die Rolle je brauchen sollte	Der Aufrufer dieser Anweisung
Lock Wait	Datenbank	Eine Anweisung, die hinter einer Sperre in der Warteschlange steht, und alle, die hinter ihr warten	Der Aufrufer; auch jeder Aufrufer, der auf dieselbe Zeile wartet
Idle in Transaction	Datenbank	Eine Transaktion, die geöffnet und dann liegen gelassen wurde und Sperren und Slots hält	Die Anwendung, die sie geöffnet hat
Connection-Lebensdauer	Pool	Eine Verbindung, die tagelang offen bleibt und ihr altes Credential oder ihren alten Zustand behält	Der Pool, der sich neu verbinden muss
Wartezeit beim Pool-Acquire	Pool	Ein Aufrufer, der ewig auf eine freie Verbindung wartet	Der Service, der die Anfrage bearbeitet

Verschiedene Rollen brauchen verschiedene Werte, weil sie verschiedene Arbeit leisten. Eine Anfrage in `billing-api` ist ein Mensch, der auf eine Seite wartet; eine Anweisung, die dort dreißig Sekunden braucht, ist ein Bug. `invoice-worker` darf legitim eine längere Anweisung ausführen, während er einen Batch aufbaut. `reporting-job` führt lange Lesezugriffe planmäßig aus, auf einem Replikat, wo ein langer Lesezugriff die Writer nicht blockiert. Ein gemeinsamer Default wäre für alle falsch, deshalb hängen die Werte an der Rolle. Der Block unten verwendet eine beispielhafte Notation, und die Werte sind hypothetisch.

```
// 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;
```

Die letzte Zeile spiegelt eine Lehre zu Migrationen wider, die Kapitel 11 behandelt: Eine Schemaänderung darf lange laufen, sollte aber nur kurz auf eine Sperre warten.

Zwei Regeln zur Reihenfolge sind wichtiger als die einzelnen Werte. Der Statement-Timeout der Datenbank sollte kürzer sein als die Request-Deadline der Anwendung, denn wenn die Anwendung zuerst aufgibt, lässt sie eine Anweisung zurück, die noch läuft, und der Slot bleibt belegt; und die Wartezeit beim Pool-Acquire sollte die kürzeste von allen sein, damit Aufrufer früh von einer Überlastung erfahren. Greift eine Deadline zuerst, kann die Verbindung noch eine laufende Anweisung tragen, sodass der Service sie schließt und nicht an den nächsten Aufrufer verleiht.

Die Idle-Transaktion ist die überraschende. Ein Service, der eine Transaktion öffnet, einen externen Zahlungsprovider aufruft und mit noch offener Transaktion wartet, lässt die Langsamkeit des Providers Sperren und einen Slot auf der Datenbank halten. Die langfristige Lösung besteht darin, externe Aufrufe außerhalb von Transaktionen zu halten, was eine Codeänderung ist; der Timeout hält die Kosten begrenzt, solange diese Änderung im Backlog liegt.

Eine Verbindung, die wochenlang lebt, behält alles, womit sie geboren wurde, sodass eine maximale Lebensdauer den Pool zwingt, seine Verbindungen regelmäßig zu ersetzen. Sie sollte kürzer sein als die Lebensdauer des Credentials, mit etwas Streuung zwischen den Verbindungen, damit nicht alle in derselben Minute neu verbinden. Kapitel 4 erklärt, warum ein abgelaufenes Credential eine bereits offene Session nicht schließt; mit der Lebensdauer schließt der Pool seine eigenen.

Defaults, die eine Session aufheben kann

Ein Timeout, der als Rollen-Default gesetzt ist, ist ein Default. Bei vielen Engines darf eine Session ihre eigenen Einstellungen ändern, darunter ihren Statement-Timeout und ihren schreibgeschützten Modus, und sie darf die Rolle wechseln. Für einen Service ist das akzeptabel: Die Limits existieren für Unfälle, und Code hebt seinen eigenen Timeout nicht versehentlich auf. Für einen Menschen mit einem SQL-Client ist es das nicht, denn die Session, die das Limit braucht, ist die Session, die es aufheben kann.

Zwei Kontrollen sitzen dort, wo die Session nicht hinreicht. Das Gateway öffnet jede menschliche Session mit angewendeten Limits und lehnt eine Anfrage ab, Einstellungen auf einer geschützten Liste zu ändern; wie das Gateway das macht, behandelt Kapitel 9. Der Preis ist, dass Clients, die solche Einstellungen beim Verbinden senden, nicht mehr funktionieren, bis jemand sie aufgelistet hat. Menschliche Sessions erhalten ihre Limits vom Gateway: Es setzt beim Anlegen des Session-Logins ein Verbindungslimit pro Login nach Art der Session und zählt, wie viele Sessions eine Person halten darf. Keines von beiden ist ein Limit pro Rolle, denn `human_jit_write` ist eine Obergrenze, deren Mitglied kein Login ist. Keines wird dem Client überlassen. Für jedes Limit, das keines von beiden hat, sollte die Tabelle der Kontrollen sagen, dass es ein Default ist und keine Garantie.

Den Aufrufer benennen

Wenn Robin sich die Verbindungen ansieht, lautet die nützliche Frage „Wem gehört diese?“, und der Login beantwortet die erste Hälfte. Eine Rolle sagt, dass eine Verbindung zu `billing-api` gehört. Sie sagt nicht, welche Instanz, welcher Release oder welcher Lauf eines geplanten Jobs. Die meisten Engines und Pooler erlauben einem Client, an seine Verbindung ein kurzes Label zu hängen, meist Anwendungsname genannt, und es erscheint im Datenbank-Log und in der Ansicht der aktuellen Aktivität.

Eine Konvention ist mehr wert als das Feature. Ich würde drei Dinge in das Label aufnehmen: den Service, die Instanz oder den Lauf und den Release. Zum Beispiel sagt `reporting-job/run-0412/v31`, welcher Job, wel-

cher nächtliche Lauf und welcher Build. Das Label macht aus „jemand hält vierzig Verbindungen“ die Aussage „der Dienstagslauf des Reports hält vierzig Verbindungen, und es ist der neue Build“.

Zwei Warnungen gehören dazu. Erstens wird das Label vom Client geliefert und kann auf beliebige Werte gesetzt werden, sodass es der Diagnose und der Zuordnung dient, nicht der Autorisierung. Der Login ist die Identität; das Label ist eine Notiz auf dem Umschlag. Zweitens teilt ein Pooler im Transaktionsmodus Server-Verbindungen zwischen Clients, sodass das Label auf der Serverseite möglicherweise nicht das des Clients ist. Manche Pooler leiten es pro Transaktion weiter, manche nicht, und das eigene Log des Poolers über seine Clients ist dann die Aufzeichnung, die den Namen trägt. Finden Sie heraus, was Ihr Setup tut, bevor Sie sich darauf verlassen.

Das Label speist auch Kapitel 14: Ein Audit-Log ist leichter zu lesen, wenn jede Zeile den Service und den Lauf trägt und nicht nur einen Login. Die Kosten sind eine Zeile im Start jedes Services und eine Konvention mehr, die zu pflegen ist, wenn ein neuer Service auftaucht.

Verbindungen sind nicht die einzige Ressource, die eine Rolle verbrauchen kann. Speicher für Sortieren und Joins, temporärer Plattenplatz, CPU und die Zahl paralleler Worker werden ebenfalls geteilt, und Engines unterscheiden sich stark darin, ob sie diese pro Rolle begrenzen können. Wo die Engine es nicht kann, muss die Isolation durch Platzierung entstehen: Ein schwerer Leser gehört auf ein Replikat, was das Argument von Kapitel 6 ist. Prüfen Sie, welche Limits Ihre Engine pro Rolle setzen kann, bevor Sie irgendjemandem eines davon versprechen.

Ledgerly: die Nacht, in der der Report die API fraß

Kehren Sie zur Dienstagnacht zurück. Der Ausgangszustand erklärt sie in einem Satz: `reporting-job` lief als `admin` gegen den Primary und konkurrierte damit unter demselben Namen mit `billing-api` um dieselben Slots, ohne Limit für den einen oder den anderen und ohne Timeout für irgendetwas.

Alex' Plan beginnt nicht mit dem Pooler. Er beginnt mit dem, was am günstigsten ist und den meisten Schutz bietet, und arbeitet die Liste weiter ab, während das Team lernt.

1. **Limits pro Rolle.** Mit den Anwendungsrollen aus Kapitel 3 erhält jede ein Verbindungslimit aus einer Tabelle wie der oben. Der Report verliert die Möglichkeit, der API sofort ihre Slots wegzunehmen, auch wenn er in der ersten Nacht noch den Primary liest.
2. **Timeouts pro Rolle.** `app_billing_api` erhält einen kurzen Statement-Timeout und ein kurzes Idle-in-Transaction-Limit, und der Report erhält einen langen Statement-Timeout, der nicht für die API gilt.
3. **Labels.** Jeder Service und jeder geplante Lauf setzt sein Label. Die nächste Seite nennt den Lauf, nicht nur die Rolle.
4. **Ein Pooler.** Die fünf Services verbinden sich über ihn, und er wird ihr einziger Zugang, wobei die Gesamtzahl pro Rolle von einer der drei oben genannten Stellen kommt.
5. **Ein Umzug auf das Replikat.** `reporting-job` liest von `ledgerly_replica_2` mit eigenem Budget, und der Primary trägt seine Arbeit nicht mehr. Kapitel 6 beschreibt den Umzug selbst.
6. **Ein Alarm vor der Ablehnung.** Das Platform-Team beobachtet die Auslastung jedes Pools, nicht nur die Ablehnungen. Ein Pool, der stundenlang bei, sagen wir, neunzig Prozent seines Limits bleibt, sagt „heben Sie das Limit an oder reparieren Sie den Aufrufer“, solange noch Zeit ist.

Auch der Job braucht eine eigene Änderung, denn ein Limit, das den Job ablehnt, ist nur nützlich, wenn der Job gut darauf reagiert, abgelehnt zu werden. Ein Job, der nach einer Ablehnung sofort einen Retry durchführt, ist die Menschenmenge an der Tür in kleinerer Form.

```
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
```

Mit diesen Maßnahmen hätte dieselbe Änderung, die die Parallelität des Reports erhöht hat, ein anderes Ergebnis. Der Job erreicht sein Limit und wird abgelehnt, er bremst sich selbst und braucht länger, und der API gehen deswegen die Slots nicht aus. Ein Job, der einen Batch aufgibt, sollte außerdem festhalten, was er übersprungen hat, damit nichts halb gebaute Aggregate als vollständig liest; das ist das eigene Design des Jobs und geht

über dieses Kapitel hinaus. Das Review, das die Änderung genehmigt hat, wäre trotzdem in Ordnung gewesen, denn das Design verlässt sich nicht mehr darauf, dass ein Reviewer das Zusammenspiel zweier Services bemerkt.

Abwägungen und Kosten

Limits pro Rolle kosten Feinabstimmung. Ein zu niedriges Limit lehnt legitime Last ab, und die Ablehnung sieht für den Service, der sie erhält, wie ein Ausfall aus. Limits brauchen einen Verantwortlichen und ein Review, wann immer ein Service skaliert oder eine neue Workload hinzukommt, denn die Summe aus Instanzen mal Pool-Größen kann das Limit überschreiten, ohne dass eine einzelne Änderung falsch aussieht. Das Platform-Team wird zur Stelle, an der diese Fragen gestellt werden, was nützlich ist und zugleich eine neue Warteschlange.

Ein Pooler ist ein neues System im Pfad. Jede Abfrage läuft nun durch ihn, und wenn er ausfällt, fällt jeder Service aus. Er muss in dem Incident verfügbar sein, in dem Sie ihn am dringendsten brauchen, also braucht er eigene Redundanz, eigenes Monitoring und ein eigenes Runbook. Im Transaktionsmodus nimmt er außerdem Session-Features weg, von denen manche Codes abhängen, und er macht das Datenbank-Log schwerer lesbar. Haben Sie einen einzelnen kleinen Service mit einer Instanz und einem Pool von zehn, genügen der Pool der Anwendung selbst und ein Limit pro Rolle.

Timeouts brechen legitime Arbeit ab. Ein zu knapp gesetzter Statement-Timeout tötet eine gute Anweisung, und ein naiver Retry wiederholt die Arbeit und den Abbruch. Services müssen den Abbruch behandeln, und bei Schreibvorgängen bedeutet das, darüber nachzudenken, ob die Operation ohne Schaden zweimal ausgeführt werden kann. Die Werte für `ledgerly_migrator` und für Datenkorrekturen (Kapitel 10) unterscheiden sich von denen der Anwendungen, was mehr Konfiguration bedeutet, die korrekt zu halten ist.

Ein Limit, das niemand beobachtet, schützt nichts. Ein Statement-Timeout, der so lang ist, dass er nie auslöst, oder ein Limit pro Rolle, das kein Alarm abdeckt, fügt der Konfiguration eine Zeile hinzu und keine

Grenze. Der Test ist, ob jemand es bemerken würde, wenn die Kontrolle nicht mehr funktioniert.

Labels sind billig und fälschbar. Sie helfen Menschen und kosten Disziplin, aber sie beweisen nichts, und ein Log, das sich für die Zuordnung auf sie stützt, stützt sich auf eine Konvention.

Wann man diese Kontrollen nicht einsetzen sollte: Fügen Sie keine Limits pro Rolle hinzu, bevor es Rollen gibt, denn ein Limit auf der gemeinsamen Rolle `admin` begrenzt nur alle auf einmal. Setzen Sie keine aggressiven Timeouts für einen Service, bevor Sie wissen, wie lange seine normalen Anweisungen dauern, denn der erste abgebrochene Rechnungslauf wird das Team lehren, die Timeouts abzuschalten. Messen Sie zuerst das Normale, und setzen Sie das Limit dann darüber.

Häufige Fehler

- **Das Server-Limit anheben, um einen Incident zu beheben.** Es ist verlockend, weil es sofort wirkt und heute nichts kostet. Es verschiebt den Fehler auf eine größere Menschenmenge und beseitigt das frühe Signal. Geben Sie jeder Rolle ein Budget, und lassen Sie das Server-Limit die Summe sein.
- **Den Pool für den Traffic bemessen, den Sie zu haben hoffen.** Mehr Verbindungen fühlen sich nach mehr Kapazität an, aber die Datenbank hat eine feste Menge nutzbarer Nebenläufigkeit. Bemessen Sie den Pool nach der Arbeit, die eine Instanz gleichzeitig leisten kann.
- **Im Pooler jeden Client auf einen Login abbilden.** Es ist die einfache Konfiguration, und sie bringt das gemeinsame Admin-Konto zurück. Behalten Sie den eigenen Login jeder Instanz, oder wählen Sie ausdrücklich einen stabilen Login pro Rolle und akzeptieren Sie, was das Log dann nicht mehr sagt.
- **Den Transaktionsmodus mit Session-Zustand nutzen.** Der Code funktionierte im Session-Modus, also testet ihn niemand im Transaktionsmodus. Kontext für Zeilenregeln, Identität für das Audit und temporäre Tabellen müssen innerhalb der Transaktion leben, oder der Modus muss für diese Rolle auf Session-Ebene bleiben.
- **Ein Satz Timeouts für jede Rolle.** Ein einziger Default ist für den Batch zu knapp und für die API zu locker, und das übliche Ergebnis ist,

dass er für alle angehoben wird. Hängen Sie die Werte an die Rolle.

- **Den Anwendungsnamen als Identität behandeln.** Das Label wird vom Client gesetzt. Verwenden Sie den Login zur Authentifizierung und das Label zur Erklärung.
- **Den Server mit Anwendungsbudgets füllen.** Wenn die Limits sich zum ganzen Server summieren, kann sich das Platform-Team im Incident nicht verbinden. Halten Sie eine Reserve außerhalb jeder Anwendungsrolle.

Zentrale Erkenntnisse

- Eine Verbindung ist eine begrenzte Ressource und ein Punkt für Richtlinien: Ein Limit, das dem Server gehört und nicht einer Rolle, lässt den ersten gierigen Aufrufer sich den Anteil aller nehmen.
- Ein Verbindungsbudget pro Rolle macht aus „die Datenbank ist für alle voll“ die Aussage „diesem Aufrufer sind die Slots ausgegangen“, was den Auswirkungsbereich eines außer Kontrolle geratenen Jobs auf den Job selbst verkleinert.
- Halten Sie eine Reserve an Verbindungen außerhalb jeder Anwendungsrolle, damit die Menschen und Werkzeuge, die das System reparieren, sich immer verbinden können.
- Der Pooler ist die Stelle, an der Services eintreten, und damit der Ort für zentrale Ablehnung und den zentralen Netzwerkübergang. Viele Pooler können die dynamischen Logins einer Rolle nicht summieren, sodass die Gesamtzahl pro Rolle von der Datenbank, vom Broker oder von einem Upstream-Login pro Rolle kommt, jeweils mit einem benannten Preis, wie der Abschnitt zum Pooler darlegt.
- Pooling im Transaktionsmodus beseitigt den Session-Zustand, sodass jeder Kontext, von dem eine Zeilenregel oder ein Audit-Eintrag abhängt, innerhalb der Transaktion gesetzt werden muss und laut fehlschlagen muss, wenn er fehlt.
- Statement-, Lock-Wait- und Idle-in-Transaction-Timeouts gehören zur Rolle, und der Statement-Timeout der Datenbank sollte vor der Deadline der Anwendung auslösen. Eine Session kann einen Default aufheben, sodass das Limit für Menschen dort durchgesetzt werden muss, wo sie es nicht ändern können.

- Anwendungsnamen machen Verbindungen erklärbar, beweisen aber nichts, sodass der Login die Identität bleibt und das Label eine Notiz.

Ein Verbindungslimit und ein Pooler bestimmen, wie viele Aufrufer den Primary erreichen, ändern aber nicht, wie viel Arbeit dort landet. Kapitel 6 macht den nächsten Schritt: die Lesezugriffe auf Replikate und andere Kopien zu verlagern, samt den Zugriffsfragen, die mit jeder Kopie kommen.

Kapitel 6 — Lesezugriffe, Replikate und Datenkopien

Stellen Sie sich einen Dienstagmorgen bei Ledgerly vor, wenige Tage vor Monatsende. Jordan aus dem Data-Team soll eine Zahl liefern: Wie viele Rechnungen wurden im vergangenen Monat ausgestellt, aufgeteilt nach Kundensegment und danach, ob bereits ein Zahlungsversuch fehlgeschlagen war? Die Analytics-Kopie wäre der naheliegende Ort für diese Frage, aber sie ist eine nächtliche Wiederherstellung, und die Wiederherstellung ist einen Tag alt. Die Frage braucht die Daten von heute. Jordan öffnet einen Desktop-Client, verbindet sich mit dem gemeinsamen Admin-Login und führt eine Abfrage aus, die `invoices`, `invoice_lines` und `payments` über den ganzen Monat verknüpft.

An der Abfrage ist nichts falsch. Es ist eine vernünftige Frage, gestellt von jemandem, der sein Handwerk versteht. Aber die Verbindung zeigt auf den Primary, weil das der Host ist, den sich der Client gemerkt hat, und die Abfrage liest mehrere große Tabellen, während `billing-api` Kunden bedient, die gleich ihre Rechnungen bezahlen wollen. Die Web-App wird langsam, Robin wird alarmiert, und Robin und Jordan brauchen zwanzig Minuten, um zu erkennen, dass der Bericht und die Verlangsamung dasselbe Problem sind. Niemand hat etwas Ungewöhnliches getan. Das Design bot einen einzigen Ort zum Lesen an, und jeder Lesezugriff ging dorthin.

Dieses Kapitel ergänzt einen Schritt des Zielzustands: Lesezugriffe gehen an Replikate, und das Reporting berührt den Primary nie. Im Fünf-Teile-Modell aus Kapitel 1 geht es um das Zusammenspiel von Grant und Pfad, und genau das bedeutet Autorisierung in diesem Buch. Der Grant bestimmt, was ein Lesender sieht, und der Pfad bestimmt, welche Datenbank der Lesende überhaupt erreichen kann. Das Kapitel behandelt auch, was aus diesem Schritt folgt: Jedes Replikat und jede andere Kopie der Daten ist ein neuer Ort, an dem der Zugriff entworfen werden muss. Die Frage lautet: Wenn Daten an mehr Orten gelesen als geschrieben werden, wel-

che Lesezugriffe gehen wohin, wer darf welche Kopie lesen, und wie verhindern Sie, dass sich die Kopien zu Türen vervielfachen, für die niemand zuständig ist?

Das Problem und warum es schwierig ist

Eine Datenbank, die Schreibzugriffe annimmt und zugleich jeden Lesezugriff bedient, erledigt zwei Aufgaben mit einem Satz von Ressourcen. Schreibzugriffe brauchen den Primary: Es gibt nur einen, er ist die maßgebliche Quelle der Wahrheit, und er muss für sie verfügbar bleiben. Lesezugriffe lassen sich wiederholen, sie lassen sich verteilen, und viele von ihnen vertragen Daten, die ein paar Sekunden alt sind. Lesezugriffe vom Primary wegzulenken schützt die Schreibzugriffe, und die Schreibzugriffe zu schützen heißt, das Hauptbuch zu schützen.

Der naheliegende Ansatz besteht darin, ein Replikat hinzuzufügen und die leseintensiven Workloads darauf zu richten. Das funktioniert etwa eine Woche lang. Danach treten drei Probleme auf, und keines davon löst das Replikat von selbst.

Das erste: Ein Replikat ist nicht der Primary. In den meisten Deployments ist die Replikation asynchron: Der Primary schreibt eine Änderung fest, und das Replikat wendet sie kurz darauf an. Die Lücke heißt **Replikationsverzögerung**. Ein Lesezugriff, der auf einem nachhinkenden Replikat landet, liefert eine wahre Antwort über eine etwas ältere Welt. Für einen Monatsbericht ist das unsichtbar. Für einen Kunden, der gerade eine Rechnung bezahlt hat und auf eine Seite schaut, die noch „fällig“ anzeigt, ist es ein Support-Ticket.

Das zweite Problem: Die Privilegien reisen mit den Daten. Ein physisches Replikat ist eine vollständige Kopie, einschließlich der Restricted-Spalten, und alles, was das Replikat lesen kann, kann lesen, was ein Lesender auf dem Primary lesen könnte, sofern die Grants nichts anderes festlegen. Der nächste Abschnitt erklärt, warum sich eine Rolle nicht allein auf einem Replikat sicherer machen lässt.

Das dritte Problem: Kopien vermehren sich. Das erste Replikat ist für die Anwendung da. Das zweite dient dem Reporting, dann der Untersuchung durch Menschen, dann dem Export-Schritt. Es folgt ein Warehouse, das nach Zeitplan oder per Change Data Capture gespeist wird. Jemand stellt

einen Snapshot in eine Staging-Datenbank wieder her. Jede Kopie war seinerzeit eine vernünftige Antwort auf ein echtes Last- oder Komfortproblem. Jede enthält auch Produktionsdaten, bringt ihre eigenen Credentials und ihren eigenen Netzwerkpfad mit und steht selten auf der Liste der Wege zu den Daten. Eine **Seitentür** ist eine Datenkopie oder ein Exportpfad, der die Zugriffskontrollen des Primary umgeht. In diesem Kapitel geht es um die Kopien, die Sie absichtlich anlegen; Kapitel 13 kommt auf die zurück, die versehentlich entstehen.

Eine Richtlinie, die besagt „keine Berichte auf dem Primary“, hilft Jordan nicht. Der Client merkt sich einen Host, der Login funktioniert auf allen, und der Primary ist derjenige, der immer frische Daten hat. Die Kontrolle muss in der Art liegen, wie Lesezugriffe weitergeleitet werden, und darin, womit sich jeder Login verbinden kann.

Den Lesepfad aufteilen

Ich würde damit beginnen, den Lesepfad als eigene Zugriffsfläche zu behandeln, mit eigenem Inventar, eigenen Rollen und eigenen Regeln, und an jeden Lesezugriff eine Frage zu stellen: Braucht dieser Lesezugriff den Primary? Er braucht ihn, wenn er etwas entscheidet, etwa einen Kontostand, bevor eine Zahlung eingezogen wird, oder wenn er einen Schreibzugriff sehen muss, der einen Moment zuvor in derselben Session erfolgte. Alles andere darf an ein Replikat gehen. Wie ein Service seine eigenen Lesezugriffe klassifiziert und weiterleitet, ist Anwendungsdesign, und dieses Buch klammert es aus.

Die Zugriffsregel ist die andere Hälfte, und sie besteht aus zwei Zeilen. Reporting-Lesezugriffe und Lesezugriffe von Menschen weichen nie auf den Primary aus: Ist das Replikat nicht verfügbar, wartet der Job oder schlägt fehl, und das Platform-Team wird benachrichtigt. Und die dafür verwendeten Logins können sich überhaupt nicht mit dem Primary verbinden, sodass ein Fehler sie nicht dorthin schicken kann. Ein nächtlicher Aggregationslauf, der zwei Stunden zu spät fertig wird, ist eine Unannehmlichkeit. Ein Lauf, der stillschweigend auf den Primary umschaltet, weil das Replikat ausgefallen war, ist genau die Art, wie der Ausgangszustand überhaupt entstanden ist.

Eine schreibgeschützte Rolle ist nicht harmlos

Lesezugriffe werden meist als Last behandelt. Sie sind aber auch das wichtigste Risiko für die Vertraulichkeit: Ein Ergebnis, das das System verlässt, ändert nichts, hinterlässt nichts, was sich zurückrollen ließe, und kann überallhin kopiert werden. Eine Rolle, die nicht schreiben kann, kann trotzdem jede Spalte lesen, für die sie einen Grant hat, und eine einzige schwere Abfrage von ihr kann den Primary lahmlegen. „Schreibgeschützt“ steht außerdem für drei verschiedene Mechanismen, die sich darin unterscheiden, ob die Session sie aufheben kann.

Mechanismus	Was er verhindert	Kann die Session ihn aufheben?
Ein Grant ohne Schreibprivilegien	Schreibzugriffe dieser Rolle	Nicht von selbst; aber eine Session, die in eine Rolle mit Schreibprivilegien wechseln kann, ist an ihn nicht mehr gebunden
Ein schreibgeschützter Transaktionsstandard	Schreibzugriffe in dieser Session, standardmäßig	Bei vielen Engines ja: Die Session kann ihn zurücksetzen oder eine Lese-Schreib-Transaktion starten
Ein physisches Replikat	Jeden Schreibzugriff auf diesem Host	Nein: Der Host lehnt Schreibzugriffe ab, und die Session hat keine Einstellung, um das zu ändern

Für Menschen ist das Replikat der stärkste der drei Mechanismen, und das ist ein Grund, warum routinemäßige Lesezugriffe von Menschen dorthin gehen. Was keiner von ihnen begrenzt, ist die Datenmenge, die eine Session lesen darf. Ein Row Cap für eine Session und Stufen für Exporte gehören zur Ausführung, und Kapitel 9 behandelt sie.

Grants reisen zum Replikat

Die Frage zum Replikat, die Sicherheitsprüfer als Erstes stellen, lautet: „Was kann jemand darauf tun?“ Bei Engines, die physisch replizieren, liegen Rollen und Grants im Katalog des Clusters, und dieser Katalog wird zusammen mit den Daten kopiert. Eine Rolle wird einmal auf dem Primary definiert und erscheint auf jedem Replikat. Ein Replikat ist schreibgeschützt, daher lässt sich auf ihm allein keine Rolle anlegen, ändern oder mit Grants versehen. Daraus folgt, dass sich eine Rolle nicht auf einem einzelnen Replikat sicherer machen lässt: Die Zugriffsfläche eines Lesenden auf einem Replikat einzuengen, erfordert entweder die Verbindungs-

schicht oder eine separate Kopie. Die Privilegien, die Sie in Kapitel 3 entworfen haben, gelten auf jedem Replikat, und welche Datenbank ein Login erreichen darf, ist eine Regel der Verbindungsschicht.

Das hat eine nützliche Seite. `human_readonly` hält Restricted-Spalten zurück und maskiert Confidential-Spalten dort, wo Kapitel 12 es vorsieht, und das tut es auf dem Replikat aus demselben Grund wie auf dem Primary. `app_reporting_job` liest auf einem Replikat ohne Zugriff auf Restricted-Spalten. Wenn die Grants mit minimalen Rechten im Sinn gebaut wurden, gibt ein Replikat einem Lesenden nicht mehr, als die Rolle erlaubt.

Es hat auch eine riskante Seite. Im Ausgangszustand akzeptieren die Replikate den gemeinsamen Admin-Login, dem jede Tabelle gehört. Wer mit diesem Login auf einem Replikat liest, sieht jede Spalte, was dieselbe Gefährdung wie auf dem Primary ist, abzüglich der Möglichkeit zu schreiben. Gerade deshalb halten viele Replikate für sicher, und das Ergebnis ist, dass auf den Replikaten der breiteste Zugriff liegt.

Jeden Login auf sein Replikat ausrichten

Im Zielzustand erhält jedes Replikat einen Zweck, und der Zweck wird dort durchgesetzt, wo Verbindungen angenommen werden. Zwei Begriffe werden von hier an in festen Bedeutungen verwendet. Eine **Netzwerkrichtlinie** legt fest, welche Hosts eine TCP-Verbindung zu einem Datenbank-Host öffnen dürfen; Kapitel 7 schreibt eine. Eine **Verbindungsrichtlinie** legt fest, welcher Login sich an welcher Datenbank authentifizieren darf, und von wo aus.

Die Verbindungsrichtlinie braucht einen Mechanismus, denn die Grants können sie nicht tragen, und die Netzwerkrichtlinie sieht keine Logins. Bei Engines, die einen haben, ist der Mechanismus die hostbasierte Zugriffsregel, die jeder Server aus seiner eigenen Konfiguration liest. Diese Regel richtet sich nach der Quelle der Verbindung und nach einem Login-Muster oder einer Rolle, und sie ist nicht Teil der replizierten Daten, sodass der Primary und jedes Replikat unterschiedliche Regeln halten können. Der folgende Block zeigt eine solche Richtlinie, geschrieben als die Regeln, die die drei Server halten würden.

```
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
```

Der Preis: Drei Regelsätze müssen jetzt mit dem Rollenschema Schritt halten, und eine Änderung, die auf einem Server ankommt und auf einem anderen nicht, ist Drift. Die Regelsätze gehören in dieselbe Pipeline, die die Rollendefinitionen anwendet (Kapitel 3), und eine Prüfung, die die Regeln jedes Servers mit der Richtlinie vergleicht, gehört in dasselbe Review. Wo eine Engine keine solche Regel kennt, fallen dieselben Entscheidungen dem Pooler und dem Gateway zu sowie der Netzwerkrichtlinie, die sie zulässt.

Die Paarung von `human_readonly` mit `ledgerly_replica_2` bedeutet, dass eine Person, die ein Problem untersucht, von demselben Replikat liest, das auch das Reporting nutzt. Das ist ein bewusster Kompromiss: ein Replikat, das zu schützen, zu überwachen und zu dimensionieren ist, im Austausch dafür, dass die Untersuchung durch Menschen mit dem nächtlichen Job um Kapazität konkurriert. Beginnen sich beide zu stören, ist ein drittes Replikat für Menschen eine vernünftige Aufteilung, und die Verbindungsrichtlinie ändert sich um einen Block.

Die Verbote auf dem Primary sind ebenso wichtig wie die Erlaubnisse auf den Replikaten. `app_reporting_job` und `app_vendor_export` haben auf dem Primary nichts zu suchen. Wenn ihre Logins sich dort nicht verbinden können, ist das Verhalten „liest manchmal den Primary“ aus dem Aus-

gangszustand nichts mehr, was ein Konfigurationsfehler zurückbringen kann. Ein Job, der auf den falschen Host zeigt, scheitert beim Verbinden, und das ist ein sichtbarer und günstiger Fehlschlag.

Die Ausnahme in der Gegenrichtung ist eine JIT- oder Break-Glass-Session. Der Login jeder menschlichen Session ist standardmäßig Mitglied von `human_readonly`, daher lässt sich die Regel für den Primary nicht gegen diese Rolle schreiben: Sie würde entweder Break-Glass abschneiden oder nichts bedeuten. Sie wird gegen die Art der Session geschrieben, die das Access-Gateway geöffnet hat. Eine Routine-Session wird an `ledgerly_replica_2` geschickt. Eine Session, die schreiben muss oder in einem Notfall lesen muss, was ein Replikat nicht schnell genug zeigen kann, wird auf dem Primary geöffnet, und der Antrag auf Zugriff wird aus dieser Session heraus gestellt (die Kapitel 7, 8 und 10 beschreiben das Modell).

Damit bleibt auf dem Primary ein Zeitfenster: Bis ein Antrag genehmigt ist, kann eine solche Session tun, was `human_readonly` tut. Drei Regeln halten das Fenster klein, und Kapitel 7 formuliert sie als Teil der Session-Arten. Nur Personen, denen die Richtlinie erlaubt, diese Art anzufordern, dürfen sie öffnen. Der Grund wird protokolliert, wenn die Session sich öffnet, und die Session hat ein kurzes Öffnungsfenster, in dem ein Antrag folgen muss. Folgt keiner, löst ein Alarm aus. Wer nur Daten ansehen will, öffnet keine solche Session; dafür ist das Replikat da, und deshalb sind Routine-Lesezugriffe auf dem Primary nicht erlaubt.

Auch das Replikat hat einen Host

Ein physisches Replikat ist eine Kopie der Datendateien, daher erreichen seine Hosts, Volumes, Snapshots und Replikations-Credentials dieselben Daten, ohne eine Datenbankrolle zu durchlaufen. Ich würde für Replikate dieselben Host-Zugriffsregeln, dieselbe Speicherverschlüsselung und dieselbe Aufbewahrung wie für den Primary anwenden und das festhalten, wenn das Replikat angelegt wird, weil später niemand daran denken wird. Kapitel 13 kommt auf Snapshots und Backups zurück.

Reporting, Analytics und die Wahl der Kopie

Replikate lösen ein Lastproblem für Lesezugriffe, die günstig und häufig sind. Reporting ist das Gegenteil: Lesezugriffe, die selten, groß und langsam sind. Die Frage ist, ob Reporting auf ein Replikat oder auf etwas anderes gehört, und die Antwort hängt davon ab, was die Berichte brauchen.

Drei Orte, von denen ein Bericht lesen kann

Option	Aktualität	Last auf dem Primary	Zugriffsfläche	Geeignet für
Ein Reporting-Replikat (`ledgerly_replica_2`)	Sekunden bis Minuten	Nur Replikation	Dieselben Daten und Rollen wie der Primary	Operative Berichte, Exporte, Untersuchungen
Eine separate Kopie, die nach Zeitplan geladen wird (`ledgerly_analytics`)	So alt wie der letzte Ladevorgang	Ein Ladevorgang pro Zeitplan	Eigene Rollen und eigene Daten	Modelle, Dashboards, Massenanalysen durch das Data-Team
Eine Kopie, die per Change Data Capture gespeist wird	Sekunden bis Minuten	Ein Leser des Änderungs-Streams	Eine neue Pipeline und ein neuer Store	Analytics nahezu in Echtzeit, viele Consumer

Ich würde operatives Reporting an ein Replikat schicken und Massenanalysen an eine separate Kopie, und ich würde mich dagegen sträuben, die Arbeit des Data-Teams auf das Replikat zu legen, aus einem Grund, der nichts mit Last zu tun hat. Ein Replikat trägt die Gestalt und die Sensibilität des Produktionsschemas. Das Data-Team braucht keine Restricted-Spalten und sieht im Zielzustand auch keine rohen Confidential-Spalten, weil die Analytics-Kopie beim Laden maskiert wird, mit Ausnahme der Anzeigehinweise, die dazu da sind, angezeigt zu werden (Kapitel 12 beschreibt, wie, und nennt die Ausnahme). Ein Replikat lässt sich nicht pro Consumer maskieren, ohne zu ändern, was das Replikat ist. Eine separate Kopie kann für ihr Publikum gebaut werden.

Deshalb wird die Analytics-Kopie in diesem Buch nirgends als Replikat bezeichnet. Sie wird nach Zeitplan vom Primary geladen, hat andere Rollen, und ihr Inhalt kann von der Quelle abweichen. Sie Replikat zu nennen

verleitet zu der Annahme, sie trage dieselben Kontrollen, und im Ausgangszustand trägt sie weniger.

Change Data Capture ist die Option, zu der ich zuletzt greifen würde. Sie fügt eine Pipeline hinzu, die laufen muss, einen Store, der abgesichert werden muss, eine eigene Verzögerung und einen Schemavertrag zwischen Producer und Consumer. Für ein Unternehmen von der Größe Ledgerlys würde ich zuerst das geplante Laden nehmen und aufschreiben, was mich umstimmen würde: zum Beispiel Berichte, die neuere Daten brauchen, als der Zeitplan erlaubt, oder ein zweites Team mit demselben Bedarf.

Die Kosten jeder Kopie

Jede Option in der Tabelle hat denselben verborgenen Posten: etwas Neues, das geschützt werden muss. Eine Kopie ist ein Ort, an dem die Daten liegen, eine Menge von Logins, die sie lesen können, ein Netzwerkpfad, eine Aufbewahrungsregel und ein Verantwortlicher. Eine Kopie ohne Verantwortlichen ist eine Seitentür, und eine Kopie mit Verantwortlichem, aber ohne Ablauf ist eine Seitentür, die länger braucht, um aufzufallen.

Es hilft, ein kurzes Register der Kopien zu führen, mit denselben Feldern für jede. Der folgende Block ist eine illustrative Notation, und es kommt darauf an, dass die Zeilen `owner` und `ends` nicht leer sein dürfen.

```
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
```

Für Ledgerly beginnt das Register mit fünf Einträgen: den beiden Replikaten, der Analytics-Kopie, den Exportdateien, die an den Buchhaltungsdienstleister gehen, und den alten Dumps im gemeinsamen Speicher. Vier davon sind gewollt, die Dumps nicht, und deshalb ist das Register nütz-

lich: Die Dumps bekommen einen Verantwortlichen und ein nahes Enddatum. Kapitel 13 verwendet dasselbe Register für die Kopien, die außerhalb der Datenbanken liegen.

> **Hinweis:** Ein Register, das niemand öffnet, ist keine Kontrolle, daher wird es an den Schritt gebunden, der Kopien erzeugt. Der Bereitstellungsschritt weigert sich, ein Replikat, eine Snapshot-Wiederherstellung, eine Staging-Datenbank oder einen Export-Job anzulegen, solange keine Zeile mit einem Verantwortlichen und, wo die Art es verlangt, einem Enddatum existiert. Das Zugriffs-Review (Kapitel 14) fragt dann, welche Kopien existieren und wer jede lesen kann.

Die Weigerung hat ihren Preis: Jemand füllt eine Zeile aus, bevor ein Team unter Termindruck seine Kopie bekommt, und in den ersten Wochen gibt es Beschwerden. Außerdem erfasst sie nur Kopien, die über den Bereitstellungsschritt entstehen, sodass eine von Hand angelegte Kopie weiterhin eine Seitentür ist, und damit befasst sich Kapitel 13.

Reichweite, nicht nur Grants

Eine Rolle, die auf dem Papier richtig ist, ist trotzdem falsch, wenn der falsche Login sich verbinden kann. Für jede Kopie hält der Grant zurück, was das Publikum nicht braucht (Kapitel 3 und 12), und die Reichweite bestimmt, welche Netzwerkpfade und Principals überhaupt eine Verbindung öffnen können. In einem flachen Netzwerk erreicht ein durchgesickerter Login eines beliebigen Services jede Kopie, und die Reichweite begrenzt, was ein Leck wert ist, wie Kapitel 4 sagt. Die Tabelle nennt die Reichweitenregel für jeden Ort; die Netzwerkrichtlinie, die die Host-Seite durchsetzt, ist die aus Kapitel 7, und die Verbindungsrichtlinie oben setzt die Login-Seite durch.

Ort	Erreichbar von	Darf nicht erreichbar sein von
`ledgerly_primary`	Pooler-Hosts, dem Gateway für JIT- und Break-Glass-Sessions, dem Pipeline-Schritt, der Migrationen anwendet	Reporting- und Export-Jobs, routinemäßigen Sessions von Menschen, der Web-Schicht direkt

Ort	Erreichbar von	Darf nicht erreichbar sein von
`ledgerly_replica_1`	Pooler-Hosts, für die Services, die Daten durchsuchen	Dem Gateway für Routine-Lesezugriffe, dem Analytics-Ladevorgang, der Web-Schicht direkt
`ledgerly_replica_2`	Pooler-Hosts für Reporting und Export, dem Gateway für routinemäßige Lesezugriffe von Menschen	Services, die die Anwendung betreiben, der Web-Schicht direkt
`ledgerly_analytics`	Dem Ladejob, `reporting-job` für seine Ergebnisse, dem Gateway für Analytics-Sessions	Den Services, die die Anwendung betreiben

Den Lesepfad von Ledgerly migrieren

Betrachten wir, wie Alex Ledgerly vom Ausgangszustand zu diesem Design bringen würde, ohne den nächtlichen Job gleich beim ersten Versuch zu brechen. Die Reihenfolge zählt mehr als die Geschwindigkeit, und ich würde in fünf Schritten vorgehen.

1. **Die Lesezugriffe inventarisieren.** Listen Sie für jeden Service die Abfragen auf, die er ausführt, und sortieren Sie sie in Lesezugriffe, die den Primary brauchen, und solche, die ihn nicht brauchen. Die meisten Listenabfragen von `billing-api` gehören zur zweiten Art. Seine Schreibzugriffe und seine Weiterleitungen nach einem Schreibzugriff nicht. Die Abfrage-Logs sind das Rohmaterial für die Routing-Regeln.
2. **Jedem Replikat einen Zweck geben.** Schreiben Sie die Verbindungsrichtlinie für `ledgerly_replica_1` und `ledgerly_replica_2` und wenden Sie sie zuerst auf die neuen Logins an; der gemeinsame Login bleibt vorerst bestehen.
3. **`reporting-job` und den Export-Schritt umziehen.** Richten Sie beide mit eigenen Logins auf `ledgerly_replica_2` aus und verbieten Sie diese Logins dann auf dem Primary. Lassen Sie beide Wege einige Zyklen lang nebeneinander laufen und vergleichen Sie die Ergebnisse, bevor Sie den alten Pfad abschalten.
4. **Die durchsuchenden Lesezugriffe umziehen.** Leiten Sie die durchsuchenden Lesezugriffe von `billing-api` und `admin-console` hinter einem Flag auf `ledgerly_replica_1` und erhöhen Sie den Anteil schrittweise.

5. **Den gemeinsamen Login auf den Replikaten schließen.** Sobald dort nichts Legitimes mehr davon Gebrauch macht, entziehen Sie ihm die Möglichkeit, sich mit einem der beiden Replikate zu verbinden. Dieser Schritt macht den Rest dauerhaft, und er ist der, den Teams hinauszögern.

Auch der Pfad des Data-Teams ändert sich. Jordans Monatsend-Frage hat jetzt zwei Antworten. Genügt eine Zahl von gestern, geht sie über `human_analytics_reader` an `ledgerly_analytics`. Braucht sie die Daten von heute, geht sie über das Access-Gateway mit `human_readonly` an `ledgerly_replica_2`, in einer Session, die protokolliert wird, die keine Restricted-Spalten sehen kann und die den Primary nicht berühren kann. Was Jordan verliert, ist die Möglichkeit, einen Desktop-Client auf einen beliebigen Host zu richten und damit zum Ziel zu kommen. Was Jordan gewinnt: Eine schwere Abfrage ist ein Problem für das Replikat und ein geringes für alle anderen.

Ein Replikat vor seinen Lesenden schützen

Ein Replikat, das Menschen und einen nächtlichen Job bedient, wird schwere Abfragen sehen, und bei manchen Engines kann eine lange Abfrage die Replikation verzögern oder abgebrochen werden, damit die Replikation weiterlaufen kann, je nach Konfiguration. Die Kontrollen sind die aus Kapitel 5, angewendet auf die Rollen des Replikats: ein kürzerer Statement-Timeout für `human_readonly` als für `app_reporting_job`, ein Verbindungslimit pro Login, das das Gateway für den Login jeder menschlichen Session setzt, und ein Export-Schritt, der außerhalb des Zeitfensters des nächtlichen Aggregationslaufs läuft. Fügen Sie einen Alarm für die Verzögerung hinzu und schicken Sie die Alarmierung an die Person, die im Register als Verantwortliche für das Replikat eingetragen ist. Ein Alarm, den niemand erhält, ist keine Kontrolle.

Abwägungen und Kosten

Ein Replikat ist ein System, das betrieben werden muss. Es muss überwacht, mit Patches versorgt, dimensioniert und wiederhergestellt werden, und für ein kleines Team genügen vielleicht ein Replikat und eine nächtli-

che Kopie. Das Zugriffsargument gilt auch ohne Replikat: Sie entscheiden, welche Logins den Primary erreichen und welche nicht.

Das gemeinsame Nutzen von `ledgerly_replica_2` hat einen Preis. Menschen und der nächtliche Job konkurrieren um die Kapazität eines Replikats, im Austausch für einen einzigen Ort, der zu schützen ist. Die Aufteilung auf ein drittes Replikat ist in Bezug auf Richtlinien günstig und im Betrieb teuer.

Die Verbindungsrichtlinie bedeutet mehr Konfiguration. Regeln pro Server müssen zum Rollenschema passen und ein Failover, einen Neuaufbau und ein neues Replikat überstehen. Drift zwischen Servern ist der Fehler, und die Vergleichsprüfung ist der Preis dafür, ihn sichtbar zu halten.

Manche Bildschirme werden nachhinken. Lesezugriffe vom Primary wegzunehmen bedeutet Verzögerung, und jemand wird es bemerken. Vereinbaren Sie mit Support und Produkt vorab, welche Bildschirme nachhinken dürfen, und prüfen Sie zuerst die Verzögerung, wenn bei einer Untersuchung auf einem Replikat etwas nicht aufgeht (Kapitel 15).

Das Register erzeugt Reibung. Eine Zeile zu verlangen, bevor eine Kopie existiert, bremst das Team, das die Kopie wollte, und diese Verlangsamung ist der beabsichtigte Effekt. Ein Register, das bei der Anlage niemand ausfüllen muss, ist eine Liste dessen, woran sich jemand erinnert hat.

Es gibt Fälle, in denen der Rat dieses Kapitels nicht greift. Ein kleines System, dessen Primary Spielraum hat und bei dem die Entwickler, die davon lesen, sich bereits über einen kontrollierten Pfad verbinden, braucht für die Last vielleicht kein Replikat, und die Trennung des Lesepfads bringt weniger. Ein System mit sehr strengen Aktualitätsanforderungen für jeden Lesezugriff muss vielleicht die meisten Lesezugriffe auf dem Primary ausführen und seine Mühe darauf verwenden, einzuschränken, was diese Lesenden tun können. Ein Replikat ist außerdem kein Backup; Kapitel 13 behandelt Backups.

Häufige Fehler

- **Ein Replikat für weniger sensibel halten als den Primary.** Es lässt sich nicht beschreiben, also sammelt es breiten Zugriff an: den gemeinsamen Login, eine dauerhafte Leserolle für alle, eine weitere Fire-

wall-Regel. Es enthält dieselben Daten, also bekommt es dieselben Grants und eine ebenso enge Verbindungsrichtlinie.

- **Reporting oder Lesezugriffe von Menschen auf den Primary ausweichen lassen.** Das Ausweichen ist verlockend, weil der Bericht weiterläuft, wenn das Replikat ausfällt. Lassen Sie den Lesezugriff stattdessen fehlschlagen und verbieten Sie diese Logins auf dem Primary, damit ein Fehler sie nicht dorthin schicken kann.
- **Lesezugriffe, die etwas entscheiden, an ein Replikat schicken.** Der Lesezugriff ist günstig und das Replikat ist untätig, also wirkt es wie ein leichter Gewinn. Eine Entscheidung auf veralteten Daten, gefolgt von einem Schreibzugriff, ist die Art, wie Kontostände und Status auseinanderdriften. Sortieren Sie Lesezugriffe danach, was auf sie folgt, nicht danach, wie schwer sie sind.
- **Ein Warehouse bauen, indem man einen Menschen auf ein Replikat richtet.** Das Data-Team trägt dann das Schema der Produktion, die Sensibilität der Produktion und die Kapazitätsgrenzen der Produktion. Geben Sie Analytics eine eigene Kopie und eigene Rollen (Kapitel 12 und 13).
- **Kopien ohne Register hinzufügen.** Jede war ein Gefallen für ein Team unter Termindruck. Sechs Monate später kann niemand sagen, wie viele es gibt. Machen Sie Verantwortlichen und Enddatum zum Teil der Anlage einer Kopie.
- **Den Host und die Snapshots vergessen.** Die Rollen stimmen, und der Volume-Snapshot des Replikats ist für jeden lesbar, der Zugriff auf die Infrastruktur hat. Nehmen Sie den Host, den Speicher und die Snapshots in die Kontrollen für jede Kopie auf.
- **Annehmen, eine Rolle lasse sich pro Replikat einengen.** Bei Engines, die physisch replizieren, gelten Rollen clusterweit. Engen Sie die Zugriffsfläche eines Lesenden in der Verbindungsschicht oder mit einer separaten Kopie ein.

Zentrale Erkenntnisse

- Schreibzugriffe brauchen den Primary, die meisten Lesezugriffe nicht; die erste Frage für den Lesepfad lautet daher, ob ein Lesezugriff etwas entscheidet oder auf einen Schreibzugriff folgt. Reporting und Lesezugriffe von Menschen weichen nie auf den Primary aus.

- „Schreibgeschützt“ sind drei Mechanismen, und nur das Replikat lässt sich von einer Session nicht aufheben; eine schreibgeschützte Rolle begrenzt also Schreibzugriffe und nicht das, was eine Session lesen darf.
- Ein Replikat enthält dieselben Daten wie der Primary und unterliegt denselben clusterweiten Rollen; die Grants aus Kapitel 3 und die Verbindungsrichtlinie entscheiden also gemeinsam, was ein Lesender sehen kann.
- Die Verbindungsrichtlinie ist eine Regel pro Server, die sich nach Quelle und Login richtet und mit dem Rollenschema Schritt hält; jedes Replikat hat einen Zweck, und der gemeinsame Admin-Login wird auf keinem von ihnen akzeptiert.
- Eine JIT- oder Break-Glass-Session auf dem Primary braucht eine Person, die sie öffnen darf, einen protokollierten Grund, ein kurzes Fenster und einen Alarm, wenn kein Antrag folgt.
- Analytics gehört auf eine separate Kopie, die für ihr Publikum gebaut ist; sie Replikat zu nennen verbirgt, dass sie andere Kontrollen trägt.
- Jede Datenkopie braucht einen Verantwortlichen, eine Leserliste und, außer bei den Replikaten und der Analytics-Kopie, ein Enddatum, und der Schritt, der Kopien erzeugt, weigert sich zu laufen, solange sie fehlen.

Der Lesepfad hat nun eigene Rollen, eigene Ziele und eine kurze Liste von Kopien, aber wir haben noch nicht gesagt, wie eine Person zu einem davon gelangt. Das nächste Kapitel behandelt die Zugangspfade für Menschen und vergleicht eine Bastion, einen Datenbank-Proxy und ein Access-Gateway.

TEIL III

Zugriff durch Menschen

Kapitel 7 — Zugangspfade: Bastionen, Proxies und Access-Gateways

Stellen Sie sich vor, Alex aus dem Platform-Team bekommt in einer Planungsrunde eine schlichte Frage gestellt: „Wenn wir morgen einen Pfad in die Produktionsdatenbank abschalten würden, welcher täte weh?“ Niemand antwortet schnell. Alex geht der Sache nach und beginnt beim Netzwerk. Das VPN gibt jedem Entwickler eine Route zum Primary. Außerdem gibt es einen Bastion-Host, den vor zwei Jahren jemand für einen Auftragnehmer eingerichtet hat, und er hat immer noch einen offenen Port. Die Desktop-Clients auf einem Dutzend Laptops halten gespeicherte Verbindungen, und eine davon ist so eingerichtet, dass sie durch die Bastion tunnelt, während eine andere direkt über das VPN geht.

Dann fragt Alex Robin, der diese Woche Rufbereitschaft hat, wie Robin nachts an die Datenbank käme. Robin nennt ohne Zögern zwei Pfade und nach kurzem Nachdenken einen dritten. Jeder Pfad funktioniert, keiner wurde entworfen, und jeder endet im selben gemeinsamen Admin-Login. Das Netzwerk-Team kann sagen, wer im VPN ist. Die Datenbank kann sagen, dass sich der Admin-Login verbunden hat. Niemand kann sagen, welche Person hinter einer bestimmten Verbindung stand, und niemand kann den Zugriff einer einzelnen Person abschalten, ohne den Zugriff aller abzuschalten.

Die Frage dieses Kapitels lautet also: Was sollte zwischen einer Person und der Datenbank stehen, damit der Zugang zum Netzwerk und der Zugang zu den Daten zwei getrennte Entscheidungen sind, an denen jeweils ein Name hängt? Dieses Kapitel ergänzt das Access-Gateway, den ersten Schritt der menschlichen Seite von Ledgerlys Zielzustand. In den fünf Teilen des Buches trägt es den Pfad und die Identität derer, die ihn betreten:

Das Netzwerk entscheidet, wer anklopfen darf, und das Gateway setzt auf jede Session einen Namen. Die Rollen, die eine Gateway-Session erhält, und wie eine Person mehr als Lesen beantragt, gehören zu Kapitel 8.

Das Problem und warum es schwierig ist

Ein Netzwerkpfad und ein Datenbankpfad sehen aus wie dasselbe, weil der erste das ist, was man braucht, bevor man den zweiten versuchen kann. Wer im VPN ist, kann eine Verbindung zum Datenbank-Port öffnen. Ob diese Person dann etwas tun darf, entscheiden ein Benutzername und ein Passwort, und im Ausgangszustand von Ledgerly ist das Passwort geteilt. Das Netzwerk hat gesagt „Sie dürfen anklopfen“, und die Datenbank konnte nicht erkennen, wer geklopft hat.

Das ist der Kern des Problems. Netzwerkkontrollen beantworten die Frage „welche Maschinen dürfen mit welchen Maschinen sprechen“, und sie tun das gut. Sie beantworten nicht „welche Person ist das, was darf sie gerade jetzt, und was hat sie getan“. Wenn eine Netzwerkregel diese Fragen ersetzen soll, versagt sie auf leise Weise: Die Regel bleibt korrekt, während der Zugriff, den sie gewährt, veraltet. Ein Entwickler, der das Team gewechselt hat, ist immer noch im VPN. Ein Auftragnehmer, dessen Auftrag geendet hat, ist immer noch in einer Gruppe, die niemand prüft.

Drei weitere Dinge machen den menschlichen Pfad schwieriger als den Pfad eines Services.

Erstens sind Menschen die vielfältigsten Clients. Ein Service verbindet sich von einem bekannten Ort mit einem bekannten Treiber. Ein Mensch nutzt vielleicht einen Kommandozeilen-Client, eine Desktop-GUI, ein Notebook, ein Browser-Tool oder ein Skript, das er letzten Dienstag geschrieben hat. Jedes hat seine eigene Art, Credentials zu speichern, seinen eigenen Standard für Auto-Commit und seine eigene Gewohnheit, Ergebnisse auf die lokale Festplatte herunterzuladen. Kapitel 1 hat die Clients aufgelistet, die das Passwort speichern und automatisch mit einem Commit abschließen; das ist kein ungewöhnliches Setup, sondern das Standardverhalten vieler Tools, die auf Bequemlichkeit hin gebaut sind.

Zweitens verbinden sich Menschen von den am wenigsten kontrollierten Orten aus. Ein Laptop reist, schläft, wird gesichert und manchmal zu Hause geteilt. Eine gespeicherte Verbindung mit einem Passwort darin wan-

dert mit dem Laptop, ebenso jedes Ergebnis, das in eine Datei exportiert wurde. Die Datenbank kann nichts davon sehen.

Drittens wird der Pfad unter Druck genutzt. Wer nachts die Datenbank braucht, hat keine Lust, ein neues Tool zu lernen, und ein Pfad, der langsam oder fragil ist, wird von jedem umgangen, der die Mittel dazu hat. Ein Design, das auf dem Papier korrekt ist und um 3 Uhr nachts gegen die alte Abkürzung verliert, hat nichts verändert, und deshalb muss die Abkürzung entfernt und nicht nur davon abgeraten werden.

Eine Richtlinie, die besagt „teilen Sie den Admin-Login nicht“, lässt sich nicht prüfen, denn das Datenbank-Log zeigt den Admin-Login und sonst nichts. Ein Design, in dem jede Session ihren eigenen Login hat, lässt sich prüfen, ebenso eine Netzwerkregel, die nur eine einzige Instanz an die Datenbank lässt. Der Rest dieses Kapitels vergleicht die Möglichkeiten, ein solches Design zu bauen.

Vier Pfade hinein im Vergleich

Vor dem Vergleich zwei Begriffe, denn die Wörter werden in der Branche locker verwendet, und das Buch braucht sie festgelegt.

Eine **Bastion** ist ein gehärteter Host, der der einzige Netzwerkpfad zu einem geschützten System ist. Sie leitet Verbindungen weiter. Sie versteht das Datenbankprotokoll nicht und kann daher ein `SELECT` nicht von einem `DROP` unterscheiden, und sie weiß nicht, wer hinter der Verbindung steht, es sei denn, die Person hat sich an der Bastion selbst angemeldet.

Ein **Access-Gateway** ist ein identitätsbewusster Vermittler, der einen Menschen authentifiziert, ein kurzlebiges Credential ausstellt und die Session aufzeichnet. Ein **Datenbank-Proxy** ist etwas anderes: ein protokollbewusster Vermittler, der Datenbankverbindungen bündelt, weiterleitet oder filtert. Ein Gateway kann auf einem Datenbank-Proxy aufbauen, aber die beiden beantworten unterschiedliche Fragen. Ein Proxy kennt das Protokoll; ein Gateway kennt die Person. Dieses Kapitel verwendet jeden Begriff nur in diesem Sinn.

Hier sind die Pfade, die ein Team üblicherweise in Betracht zieht, mit dem, was jeder über die Person weiß und was er hinterlässt. Die ersten vier setzen eine Person in eine Session; die letzten beiden halten die Per-

son aus einer heraus.

Pfad	Was er über die Person weiß	Was das Datenbank-Log zeigt	Hauptkosten
Öffentlicher Endpoint	Nichts über den Datenbank-Login hinaus	Ein Login, von einer beliebigen Adresse	Setzt den Datenbank-Port dem ganzen Internet aus
VPN zur Datenbank	Dass die Person im Netzwerk ist	Der gemeinsame oder personenbezogene Login	Netzwerkmitgliedschaft wird als Identität behandelt
Bastion	Wer sich am Bastion-Host angemeldet hat, wenn die Konten personenbezogen sind	Der Login, den die Person eingetippt hat, von der Adresse der Bastion	Ein weiterer Host, der mit Patches versorgt werden muss, und er sieht keine Anweisungen
Access-Gateway	Identität aus dem Single Sign-on des Unternehmens	Ein Session-Login, der auf genau eine Person und eine Session abgebildet wird	Ein neues System, das verfügbar sein muss, wenn jemand es braucht
Produktpfad, kein SQL	Die Rolle der Person in einem internen Tool	Der eigene Login des Services, mit der Person im Prüfpfad des Tools	Für jede Aufgabe muss ein Feature gebaut werden; freie Fragen haben keinen Pfad
Einreichen-und-ausführen-Executor	Die Person, die eingereicht hat, und der Reviewer, der genehmigt hat	Der Login des Executors, der eine geprüfte Menge von Anweisungen ausführt	Jede Aufgabe wartet auf ein Review; nichts ist interaktiv

Die letzten beiden Zeilen sind überhaupt keine Wege, eine Datenbank zu erreichen. Die Person hält nie eine Session, und die Anweisung läuft von einer Komponente aus, die sie zuvor geprüft hat. Sie gehören in den Vergleich, weil sie für viele Aufgaben die bessere Antwort sind: Wer nie eine Session öffnet, kann keine offen lassen. Kapitel 1 hat das Spektrum solcher Patterns eingeführt, Kapitel 9 behandelt, wie das Gateway eine Anweisung prüft, und Kapitel 10 behandelt die Patterns für Datenkorrekturen; dieses Kapitel behandelt die Sessions, die übrig bleiben, nachdem die Aufgaben, die zu diesen Patterns passen, ausgezogen sind.

Der öffentliche Endpoint

Ich würde die Datenbank für die Nutzung durch Menschen nicht dem öffentlichen Internet aussetzen, und das Argument ist kurz. Ein erreichbarer Port zieht automatisierten Verkehr an, der mit Ledgerlys Leuten nichts zu

tun hat, und jedes Credential für diese Datenbank wird dann zu etwas, das zu erraten sich lohnt. Die einzige verbleibende Verteidigung ist die Stärke des Logins, und die Prämisse des Buches ist, dass Logins durchsickern.

Es gibt enge Fälle, in denen eine verwaltete Datenbank nur öffentlich erreichbar ist und der Anbieter eine Allowlist von Quelladressen bietet. Wenn Sie in dieser Lage sind, behandeln Sie die Allowlist als Minimum und wechseln Sie auf einen privaten Pfad, sobald der Anbieter es zulässt. Der Test ist einfach: Könnte jemand, der nie für Sie gearbeitet hat, eine Verbindung zum Port öffnen? Wenn ja, beginnt jede andere Kontrolle in diesem Buch von einer schlechteren Ausgangslage.

Das VPN

Ein VPN ist eine vernünftige Netzwerkschicht. Es beschränkt den Kreis derer, die die Datenbank erreichen können, auf Personen, die eine Route erhalten haben, und die meisten Unternehmen haben bereits eines. Seine Grenze ist das, was es beweist. Es beweist, dass eine Person oder der Laptop einer Person die Prüfungen des VPN bestanden hat. Es sagt nicht, welchen Datenbank-Login sie danach verwendet haben, und es zeichnet keine Anweisungen auf.

Im Ausgangszustand von Ledgerly ist das VPN die gesamte Kontrolle. Alle darin erreichen den Primary, und der Admin-Login ist für alle derselbe. Ich würde das VPN behalten, ihm aber eine kleinere Aufgabe geben: Es bringt Menschen zum Gateway und zu den anderen internen Tools, und es bringt sie nicht mehr zum Datenbank-Port.

Die Bastion

Eine Bastion fügt eine nützliche Eigenschaft hinzu: einen einzigen Ort, an dem die Netzwerktür geschlossen werden kann. Die Firewall der Datenbank lässt nur die Adresse der Bastion zu, sodass das Abschalten eines Bastion-Kontos den Netzwerkpfad einer Person abschaltet. Hat die Bastion personenbezogene Konten und zeichnet sie auf, wer sich bei ihr angemeldet hat, erhalten Sie eine grobe Aufzeichnung darüber, wer wann verbunden war.

Was eine Bastion nicht liefert, ist alles unterhalb der Netzwerkschicht. Sobald eine Person einen offenen Tunnel durch sie hat, ist die Verbindung zur Datenbank das, was der Client der Person daraus macht. Der Login wird weiterhin von der Person getippt oder in ihrem Client gespeichert. Die Anweisungen sind für die Bastion unsichtbar. Eine Bastion räumt also das Netzwerkbild auf, ohne das Identitätsbild zu reparieren, und deshalb kann ein Team eine haben und trotzdem „wer hat diese Anweisung ausgeführt?“ nicht beantworten.

Eine Bastion hat außerdem ein stilles Wartungsproblem. Sie ist ein Host mit einer Shell, also braucht sie Patches, Schlüsselverwaltung und einen Verantwortlichen, der sich daran erinnert, dass es sie gibt. Der Host des Auftragnehmers aus dem Einstieg ist die übliche Gestalt dieses Versagens: Er war an dem Tag korrekt, an dem er gebaut wurde, und danach wurde ihm niemand zugewiesen.

Das Access-Gateway

Das Access-Gateway ist der Ort, an den ich Menschen schicken würde, aus den folgenden Gründen.

Es authentifiziert die Person gegen das Single Sign-on des Unternehmens, sodass die Identität dieselbe ist, die das Unternehmen ohnehin kontrolliert. Das Ausscheiden aus dem Unternehmen, ein Teamwechsel und eine fehlgeschlagene Multi-Faktor-Prüfung wirken am Gateway, ohne dass jemand die Datenbank bearbeitet. Es stellt ein Credential aus, das zu einer Session gehört und mit ihr stirbt, sodass es kein Passwort zu speichern gibt. Und es zeichnet die Session auf, sodass „wer hat diese Anweisung ausgeführt?“ an zwei Stellen eine Antwort hat: im Session-Eintrag des Gateways und im Datenbank-Log, das einen Login nennt, der auf genau eine Person abgebildet wird.

Gateways gibt es in zwei groben Formen. Manche sitzen im Datenpfad: Der Client der Person verbindet sich mit dem Gateway, und das Gateway öffnet die Verbindung zur Datenbank im Auftrag der Person. Andere vermitteln nur das Credential und treten beiseite, sodass der Client sich direkt mit dem kurzlebigen Login mit der Datenbank verbindet. Die erste Form kann Anweisungen aufzeichnen und Regeln auf der Verbindung durchsetzen. Die zweite ist einfacher und stellt weniger Dinge in den Pfad,

aber dann muss das Netzwerk Menschen den Zugang zum Datenbank-Port erlauben, was einen Teil des Problems zurückbringt, mit dem dieses Kapitel begonnen hat.

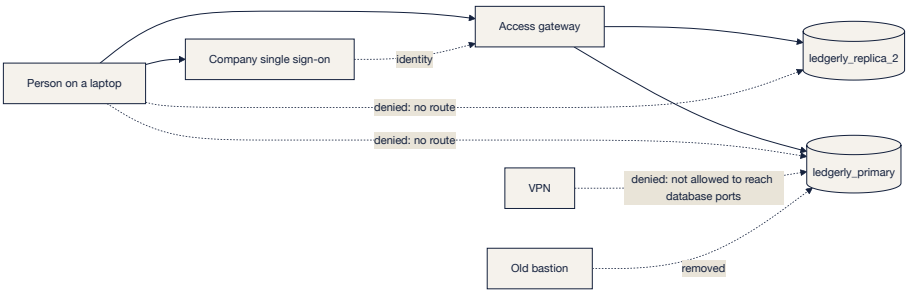
Für Ledgerly würde ich die Form im Datenpfad wählen, weil sie die Netzwerkregel absolut machen lässt: Nur die Adresse des Gateways darf die Datenbank-Hosts erreichen. Der Preis ist, dass das Gateway jede Anweisung von Menschen verarbeitet, sodass seine Kapazität und Verfügbarkeit zählen. Darauf komme ich weiter unten zurück.

Netzwerkreichweite vom Datenbankzugriff trennen

Die nützlichste Änderung in diesem Kapitel ist kein Produkt. Es ist die Trennung zweier Fragen, die der Ausgangszustand gemeinsam beantwortet.

Die erste Frage ist die **Netzwerkreichweite**: Welche Maschinen dürfen eine Verbindung zu welchem Datenbank-Host und Port öffnen. Die zweite ist der **Datenbankzugriff**: Welcher Login darf was, sobald er verbunden ist. Im Ausgangszustand beantwortet das VPN die erste und ein gemeinsames Passwort die zweite. Nach der Änderung beantwortet eine Netzwerkregel die erste für alle auf einmal, und das Gateway beantwortet die zweite für jede Person einzeln.

Das Diagramm zeigt die Zielanordnung für Menschen. Achten Sie auf die verweigerten Pfeile: Das sind die Pfade, die heute existieren und danach nicht mehr existieren werden.



Hinter diesem Diagramm stehen zwei Durchsetzungspunkte. Die Netzwerkregel auf den Datenbank-Hosts lässt Verbindungen nur vom Gateway zu, aus den eigenen Netzwerken der Services, von der Pipeline und vom

Credential-Broker. Und das Gateway leitet eine Session nur an die Datenbank weiter, die die Art der Session erlaubt: Eine routinemäßige Lese-Session geht an `ledgerly_replica_2`, und eine Session, die geöffnet wurde, um einen JIT- oder Break-Glass-Antrag zu stellen, geht an `ledgerly_primary`, wie Kapitel 6 dargelegt hat.

Ich würde die Netzwerkregel als kurze Liste aufschreiben, die jeder lesen kann, denn diese Liste ist das, was ein Zugriffs-Review prüft. Kapitel 6 hat den Unterschied definiert: Diese Liste ist die Netzwerkrichtlinie, die entscheidet, welche Hosts sich überhaupt verbinden dürfen, während die Verbindungsrichtlinie entscheidet, welcher Login sich an welcher Datenbank authentifizieren darf. Enthält sie einen Eintrag, den niemand erklären kann, ist dieser Eintrag ein Pfad.

```
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
```

Dieselbe Liste geht weiter mit dem, was die Datenbank-Hosts verlässt, und mit dem, was das Gateway erreichen kann.

```
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
```

Beachten Sie, wie wenig das VPN in diesem Modell zu tun hat. Es bewacht die Datenbank nicht mehr. Es bringt Menschen zum Gateway, und das Gateway entscheidet den Rest. Das VPN wird weniger wichtig; der Zugriff jeder Person ist jetzt an eine benannte Session gebunden.

Zwei Einträge der Liste vergisst man leicht. Zuerst der Egress. Ein Datenbank-Host, der ausgehende Verbindungen überallhin öffnen darf, lässt sich zum Hinausschicken von Daten missbrauchen, und eine kurze Allowlist (Replikation, das Audit-Log, Backups) macht daraus ein sichtbares Ereignis. Zweitens die eigene Gefährdung des Gateways: Es ist jetzt die eine erreichbare Tür, steht also selbst auf einer kurzen Liste, und es darf nicht von Orten erreichbar sein, die das VPN nicht ohnehin erreicht.

Die Liste setzt außerdem voraus, dass das Netzwerk die Aufrufer unterscheiden kann. In einem gemeinsamen Cluster-Netzwerk, in dem mehrere Services aus demselben Adressbereich kommen, kann es das nicht: Eine Regel, die „die Netzwerke der Services“ zulässt, lässt alle zu. In diesem Fall hält die Netzwerkregel alles fern, was weder ein Service noch das Gateway ist, und es sind der Pooler und der Login, die einen Service vom anderen trennen (Kapitel 5 und 6). Ich würde das als Regel für den Cluster aufschreiben, damit niemand die Netzwerkliste für präziser hält, als sie ist. Welcher Login sich an welcher Datenbank authentifizieren darf, ist die Verbindungsrichtlinie aus Kapitel 6, und dieses Kapitel verweist nur darauf.

Das Access-Gateway bei Ledgerly

Hier steht, was das Gateway in welcher Reihenfolge tut und was geschieht, wenn ein Schritt fehlschlägt. Diesen Teil würde ich sorgfältig testen, denn ein Gateway, das offen fehlschlägt, wird zum neuen gemeinsamen Passwort mit besserem Branding.

Was eine Session ist

Eine Session beginnt, wenn sich eine Person authentifiziert. Das Gateway prüft die Identität, entscheidet, welche Art von Session die Person beantragen darf (eine routinemäßige Lese-Session für die meisten; JIT- und Break-Glass-Arten nur über die Anträge aus Kapitel 8), und legt dann einen **Session-Login** an, das Konto, das die Session der Person verwendet. Er wird auf `ledgerly_primary` angelegt und repliziert (Kapitel 6). Die Session verbindet sich dann mit der Datenbank, zu der ihre Art gehört. Dieser Login trägt die Identität der Person in seinem Namen, sodass er im Datenbank-Log erscheint, und er ist standardmäßig Mitglied von `human_readonly`. Das Credential dafür wird für diese Session erzeugt, dem Gateway übergeben und der Person nie angezeigt, sodass es nichts zu speichern gibt.

Ein Login-Name hat eine feste Gestalt, damit sich eine Log-Zeile ohne Nachschlagen lesen lässt: ein Präfix, das sagt, dass er ein Session-Login ist, die Kurzkennung der Person und die eigene Nummer der Session, wie in `sess_robin_0412`. Der Name identifiziert; der Eintrag des Gateways ist es, der ihn auf die Person und den Antrag abbildet, und nichts sollte den Namen parsen, um zu entscheiden, was die Session darf.

Um Logins anzulegen und zu entfernen und um Rollenmitgliedschaften zu vergeben und zu widerrufen, braucht das Gateway Privilegien in der Datenbank. Es erhält sie über seinen eigenen Login für die Rolle `ledgerly_access_admin`. Der Pipeline-Schritt für Rollendefinitionen und der Credential-Broker nutzen diese Rolle ebenfalls, jeweils über einen eigenen Login (Kapitel 3 und 4). Damit ist der Login des Gateways ein sensibles Credential: Wer ihn hält, kann innerhalb der Rollen, die es verwaltet, alles vergeben. Ich würde es am selben Ort und unter denselben Regeln aufbewahren wie die Credentials des Eigentümers, ihm keine Datenprivilegien geben und jede Verwendung auditieren (Kapitel 14). Es der Bequemlichkeit halber in Skripte oder in einen Chat zu kopieren würde das gemeinsame Passwort eine Ebene höher neu erschaffen.

Wer welche Art von Session öffnen darf

`mayRequestKind` ist Richtlinie, und ich würde sie als Daten halten, die ein Reviewer lesen kann, nicht als Bedingung, die im Code des Gateways vergraben ist. Eine Tabelle beantwortet sie: die Single-Sign-on-Gruppe, die

Arten, die diese Gruppe öffnen darf, die jeweils längste Dauer, die Datenbank, auf der sie landet, ihre Standardwerte (schreibgeschützt, Statement-Timeout, Row Cap) und ob sie exportieren darf.

```
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
```

Die auf den Primary ausgerichteten Arten sind die sensiblen, denn eine Session auf dem Primary ist einen Antrag vom Schreibzugriff entfernt. Vier Regeln halten sie selten: Nur benannte Gruppen dürfen eine öffnen; sie muss mit einem protokollierten Grund geöffnet werden; das Gateway wartet nur kurz, sagen wir fünfzehn Minuten, darauf, dass aus ihr ein Antrag gestellt wird; und eine Session, die ohne Antrag endet, löst einen Alarm aus, denn „den Primary geöffnet und nichts getan“ ist ein Muster, das Beachtung verdient. Die Obergrenze von zwei Stunden gilt nur bis zu einem Grant: Wenn `grantJitAccess` oder `declareBreakGlass` aus Kapitel 8 einen Antrag genehmigt, setzt das Gateway das `ends_at` dieser Session auf das `expires_at` des Antrags zurück, und `endSession` weiter unten beendet die Session zu diesem Zeitpunkt. Kapitel 8 beginnt von der anderen Seite, beim Antrag.

Die Reihenfolge der Operationen

Das Gateway, das Audit-Log, die Datenbank und das Single Sign-on sind getrennte Systeme. Keine Transaktion umspannt sie, daher müssen die Reihenfolge der Schritte und der Umgang mit einer halb fertigen Session entworfen werden.

```
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))
```

Lesen Sie den Block auf seine Fehlerbehandlung hin. Der Eintrag wird geschrieben, bevor der Login existiert, sodass ein Absturz mittendrin entweder einen Eintrag mit „opening“ und keinen Login hinterlässt oder einen Login, den der Eintrag erklärt. Der Audit-Eintrag kommt vor dem Login: Ist der Log-Store nicht erreichbar, öffnet sich keine Session. Diese Entscheidung hat einen Preis, den der Abschnitt über Abwägungen nennt. Bei einer routinemäßigen Lese-Session wartet das Gateway, bis `ledgerly_replica_2` den Login erhalten hat, weil die Replikationsverzögerung sonst gelegentlich die erste Verbindung fehlschlagen ließe. Schlägt das Anlegen des Logins oder das Vergeben der Rolle fehl, versucht das

Gateway aufzuräumen und verweigert die Session. Scheitert sogar das Aufräumen, existiert ein Login ohne Privilegien, und der Sweep weiter unten entfernt ihn.

Eine Session beenden

Eine Session endet auf einem von drei Wegen: Die Person trennt die Verbindung, das Leerlauflimit läuft ab, oder `ends_at` verstreicht. Die letzten beiden zählen, weil Menschen Tabs offen lassen. Eine offene Verbindung überlebt ein abgelaufenes Credential (Kapitel 4), daher schließt das Gateway die Verbindungen selbst und entfernt dann den Login. Dieselbe Funktion bedient eine Person, die das Unternehmen verlässt: Ein Deprovisioning-Ereignis aus dem Single Sign-on ruft `endSession` auf, und die maximale Session-Dauer ist die Schranke für den Fall, dass dieses Ereignis nie eintrifft.

```
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 })
```

Hier hängt `auditOrQueue` den Eintrag an das Audit-Log an und stellt ihn, wenn das Log nicht erreichbar ist, für einen erneuten Versuch in eine Warteschlange und benachrichtigt das Platform-Team. Der Widerruf und das Entfernen sind zu diesem Zeitpunkt bereits geschehen, sodass ein Ausfall des Logs den Eintrag verzögert und nie die Entziehung des Zugriffs. Ein Login, der schon weg ist, gilt als Erfolg: Eine Session, die der Ablaufpfad aus Kapitel 8 früher beendet hat, wird hier sauber abgeschlossen, ihr Eintrag wird als beendet markiert, und es wird kein Alarm ausgelöst.

Der Sweep ruft `endSession` für jede Session auf, die schon vorbei sein sollte.

```
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")
```

Auf die Reihenfolge kommt es an. Die Grants eines Antrags werden anhand seines gespeicherten Eintrags widerrufen und erst als widerrufen markiert, nachdem der Widerruf funktioniert hat, und erst dann wird der Login entfernt, weil manche Engines sich weigern, einen Login zu löschen, der noch Privilegien hält. Ein fehlgeschlagener Widerruf oder ein fehlgeschlagenes Entfernen benachrichtigt das Platform-Team und lässt den Eintrag für den nächsten Sweep stehen; ein fehlgeschlagener Audit-Schreibvorgang wird in die Warteschlange gestellt und hält die Schleife nicht an. Der Sweep gewährt einer Session, die noch „opening“ ist, eine Schonfrist, damit er keinen Login entfernt, der erst Sekunden alt ist. Die Drift-Prüfung aus Kapitel 3, die die vorhandenen Logins mit den Einträgen des Gateways und die Grants mit den gespeicherten Tripeln vergleicht, läuft nach eigenem Zeitplan, sodass ein angehaltener Job den anderen nicht verbirgt.

Welche Datenbank, welche Rolle

Jede Session landet dort, wohin ihre Art gehört. Eine routinemäßige Session ist ein Login der Rolle `human_readonly`, der sich mit `ledgerly_replica_2` verbindet. Eine Session, die geöffnet wurde, um ei-

nen JIT- oder Break-Glass-Antrag zu stellen, wird auf `ledgerly_primary` geöffnet, und Kapitel 8 zeigt, wie der Antrag aus dieser Session gestellt wird und wie die Grants mit ihr enden. Das Gateway setzt die Zielregel durch, die Kapitel 6 als Verbindungsrichtlinie geschrieben hat: Das Gateway kennt die Art, und der Datenbank-Host weist einen Login zurück, der für eine andere bestimmt war.

Eine Person kann zwei Sessions gleichzeitig offen haben, eine routinemäßige und eine mit einem Antrag. Ich würde das möglich lassen und die Sessions sichtbar getrennt halten, damit niemand die privilegierte für eine beiläufige Abfrage benutzt. Die menschlichen Kosten sind eine zweite Verbindung im Client der Person. Der Nutzen ist, dass die privilegierte Session leicht zu finden, kurz und leicht zu beenden ist.

Was das Gateway aufzeichnet

Weil Ledgerlys Gateway im Datenpfad sitzt, kann es jede Anweisung aufzeichnen, die eine Session sendet, samt Beginn, Ende und betroffenen Zeilen. Ich würde diese Aufzeichnung dort speichern, wo die beschriebenen Personen sie nicht bearbeiten können; Kapitel 14 behandelt, wo. Ein Eintrag sieht so aus:

```
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
```

Regelwerke für Zahlungsdaten und Finanzberichterstattung fragen tendenziell, wer Produktionsdaten auf welchem Weg berührt hat; der Session-Eintrag ist diese Antwort, und die Auslegung liegt bei den Rechts- und Compliance-Verantwortlichen der Lesenden. Zwei Grenzen sind wissenswert. Ein Anweisungs-Log sagt Ihnen, was gefragt wurde, nicht was zurückgegeben wurde, und ich würde keine vollständigen Ergebnismengen speichern, weil sie das Log in eine Kopie der Daten verwandeln würden. Aus demselben Grund trägt der Fingerprint die Gestalt der Anweisung und

einen Hash ihrer Parameter statt ihrer Werte. Und ein Log, das niemand liest, zeichnet auf, ohne etwas zu schützen; legen Sie also fest, wer es ansieht und wann: Eine wöchentliche Stichprobe von Sessions, die einen JIT-Antrag hielten, ist ein Anfang, und das Zugriffs-Review ist der andere Anlass.

Kapazität und das Verbindungsbudget

Jede menschliche Session ist auch eine Datenbankverbindung, und das Budget aus Kapitel 5 hat Menschen einen festen Anteil gegeben, sagen wir als Illustration zwanzig Verbindungen. Routinemäßige Sessions von Entwicklern landen alle auf `ledgerly_replica_2`, daher ist dieses Replikat der Host des Slots für Menschen, und sein Verbindungslimit ist der Ort, an dem die Zahl schließlich durchgesetzt wird. Das Gateway sollte diesen Anteil selbst durchsetzen: ein Limit pro Person, ein Limit pro Art von Session und eine klare Ablehnung, wenn das Budget aufgebraucht ist. Eine Ablehnung am Gateway ist eine Nachricht an einen Menschen. Eine Ablehnung an der Datenbank, wenn das Budget erschöpft ist, kann einen Service verdrängen. Das Limit am Gateway zu setzen verhindert, dass das Budget für Menschen zum Problem des Services wird.

Der Laptop und die lokale Kopie

Das Gateway kontrolliert den Pfad zur Datenbank. Es kontrolliert nicht, was mit den Daten geschieht, sobald sie auf einem Laptop angekommen sind, und dort sitzt ein stiller Teil des Risikos.

Zwei Dinge bleiben auf dem Laptop. Das eine ist das Verbindungs-Setup: der gespeicherte Host, der gespeicherte Login und im Ausgangszustand das gespeicherte Passwort. Das andere ist alles, was die Person mit einem Ergebnis tut: eine exportierte Datei, eine eingefügte Tabelle, eine lokale Kopie von ein paar tausend Zeilen zum „Herumspielen“. Jedes ist eine Datenkopie im Sinne von Kapitel 6, ein Ort, an dem Produktionsdaten liegen, der keinen Verantwortlichen und keinen Ablauf hat.

Ich würde das in drei Schichten angehen, von der günstigsten zur teuersten.

Entfernen, was der Client sich merken kann. Weil Session-Credentials für eine Session erzeugt und nie angezeigt werden, hat ein Desktop-Client kein Passwort zum Speichern. Die gespeicherte Verbindung enthält nun eine Gateway-Adresse und das Single Sign-on der Person, und die alten gespeicherten Einträge für den Primary können während der Migration gelöscht werden, mit einer kurzen Liste dessen, was wo zu löschen ist. Diese Schicht kostet fast nichts und beseitigt das häufigste Leck.

Standardwerte wählen, die zur Aufgabe passen. Das Gateway kann Sessions mit schreibgeschützten Standardwerten und einem Statement-Timeout aus Kapitel 5 öffnen. Es kann auch die Zeilen begrenzen, die eine Anweisung zurückgibt: Es zählt Zeilen, während es sie streamt, schließt den Cursor an der Grenze, informiert die Person und schreibt das Ereignis in den Anweisungseintrag; eine Session, die die Grenze wiederholt erreicht, ist das, was die wöchentliche Stichprobe zuerst ansieht.

Standardwerte, die die Person aufheben kann, bleiben keine Standardwerte. Ein Client kann ein `SET` senden, das den Timeout erhöht oder den Schreibschutz abschaltet, daher führt das Gateway eine Liste von Guarded Settings und verweigert jede Anweisung, die sie ändert. Der Preis ist, dass manche Clients solche Anweisungen beim Verbinden senden und dann nicht mehr funktionieren; das ist ein Preis, den man in der Migration benennen muss, und Kapitel 9 behandelt, wie das Gateway entscheidet, welche Anweisungen es verweigert. Clients mit Auto-Commit sind weniger gefährlich, wenn die Session keine Schreibprivilegien hat, mit denen sie einen Commit ausführen könnte. Eine browserbasierte Konsole, die das Gateway selbst betreibt, geht einen Schritt weiter, denn es wird nichts heruntergeladen, es sei denn, es gibt einen Export-Button und er wird protokolliert. Diese Konsole ist langsamer und weniger angenehm als ein nativer Client, und ich würde sie als Option für Menschen anbieten, die nur schauen und nie eigene Tools brauchen, nicht als Pflicht.

Ausdrücklich entscheiden, was hinausgeht. Ob ein Ergebnis aus einer Session exportiert werden darf, ist eine eigene Regel, und sie gehört zur Maskierung und Minimierung aus Kapitel 12 und zu den Exportpfaden aus Kapitel 13. Das Gateway kann die Regel anwenden, aber es kann sie nicht überflüssig machen. Eine Person kann einen Bildschirm lesen, ihn fotografieren oder einen Wert abtippen, und ich kenne dafür keine kosteneffizien-

te Kontrolle. Das Gateway verringert die Datenmenge, die versehentlich hinausgeht, und macht bewusste Exporte sichtbar; es macht das Hinausgehen nicht unmöglich.

> **Hinweis:** Wenn Ihre Entwickler große Extrakte für Analysen brauchen, ist die Antwort meist nicht ein größeres Exportlimit am Gateway. Es ist eine maskierte Analytics-Kopie, die für diesen Zweck entworfen wurde und die die Kapitel 6 und 12 beschreiben. Das Gateway ist das falsche Werkzeug für Massendatenbewegungen.

Eine Tür, ein Single Point of Failure

Ein einziger Einstiegspunkt ist leicht zu erklären, leicht zu protokollieren und leicht zu schließen. Er ist auch ein Single Point of Failure, und das sollte man deutlich sagen, bevor man ihn empfiehlt. Ist das Gateway ausgefallen, kann niemand die Datenbank durch die Vordertür erreichen, auch nicht die diensthabende Person, die die Datenbank am dringendsten braucht, wenn etwas anderes bereits kaputt ist.

Es gibt keine Antwort, die diese Kosten beseitigt. Es gibt Antworten, die sie an einen Ort verschieben, an dem Sie mit ihnen umgehen können.

Halten Sie das Gateway klein. Es authentifiziert, legt Logins an, leitet weiter und zeichnet auf. Jedes zusätzliche Feature ist ein weiterer Grund, warum es ausfallen kann. Ich würde der Versuchung widerstehen, Ergebnistransformation und eine Genehmigungsoberfläche in denselben Service zu legen, und sie als getrennte Komponenten betreiben, die das Gateway aufruft, von denen es aber nicht abhängt, um eine Lese-Session zu starten. Wo eine Anweisung vor der Ausführung geprüft wird, ist auch diese Prüfung eine eigene Komponente (Kapitel 9).

Betreiben Sie mehr als eine Instanz und legen Sie den Zustand an einen dauerhaften Ort. Zwei Instanzen hinter derselben Adresse, die sich die Session-Einträge teilen, sorgen dafür, dass ein Neustart oder ein fehlerhaftes Deployment nicht jede Session beendet. Der Session-Store und das Audit-Log sind ebenfalls Abhängigkeiten, und ihre Verfügbarkeit ist Teil der Verfügbarkeit des Gateways. Das ist der Grund, warum die obige Funktion zum Öffnen eine Session verweigert, wenn das Log nicht erreich-

bar ist: Es ist eine bewusste Entscheidung, die den Zugriff von Menschen an das Log bindet, und es sollte eine Entscheidung sein, die das Team absichtlich getroffen und aufgeschrieben hat.

Entscheiden Sie, was die degradierten Modi sind. Es gibt zwei Ausfälle, für die man planen muss. Der eine ist ein Gateway, das ausgefallen ist, während die Datenbanken in Ordnung sind. Der andere ist ein ausgefallenes Audit-Log, während das Gateway arbeitet. Für jeden muss eine verantwortliche Person vorab entschieden haben, ob der Zugriff von Menschen aufhört oder auf einer engeren Route weitergeht und was in der Zwischenzeit aufgezeichnet wird. Die Entscheidung des Buches für einen Gateway-Ausfall lautet, dass es keine zweite Tür gibt: Der Zugriff von Menschen hört auf, Break-Glass eingeschlossen, denn eine Break-Glass-Session ist eine Gateway-Session mit einer gewichtigeren Rolle. Der Weg, das selten zu halten, ist die oben genannte Redundanz, nicht ein Notfall-Login außerhalb des Gateways. Kapitel 8 gibt die Begründung und sagt, was ein Team tut, wenn es anders entscheidet, und Kapitel 15 beschreibt, wie man den Ausfall übt. Bei einem Ausfall des Audit-Logs dürfen Lesezugriffe auf einem Replikat Einträge puffern, Schreibzugriffe nicht (siehe die Abwägungen unten).

Behandeln Sie das Gateway wie Produktion. Es hat einen Verantwortlichen, einen Rufbereitschaftspfad, einen Health-Check, den ein Mensch lesen kann, und einen Deployment-Prozess mit Rollback. Das Platform-Team ist dafür verantwortlich. Ist das Gateway das Nebenprojekt eines einzelnen Entwicklers, dann ist der Single Point of Failure dieser Entwickler.

Das Gateway bündelt außerdem ein anderes Risiko: Eine Kompromittierung des Gateways ist eine Kompromittierung des Zugriffs auf alles, was es erreichen kann. Deshalb wird sein Login `ledgerly_access_admin` so geschützt, wie es weiter oben beschrieben ist, und es ist kein Grund, das Gateway zu meiden: Der Ausgangszustand hat ein gemeinsames Passwort, an das gesamte Team kopiert, in derselben Position.

Ledgerlys Weg von vielen Pfaden zu einem

Ein Team in Ledgerlys Lage würde nicht alles an einem Tag umschalten. Ich würde die Schritte in einer Reihenfolge gehen, die an jedem Punkt einen funktionierenden Pfad verfügbar hält.

1. **Die Pfade inventarisieren.** Listen Sie jeden Weg auf, auf dem eine Person heute eine Ledgerly-Datenbank erreichen kann: die VPN-Routen, die Bastion, die gespeicherten Client-Einträge, alle Skripte, die sich mit dem Admin-Login verbinden. Alex' Rundgang im Einstieg war dieser Schritt, und er sollte in einer schriftlichen Liste enden, die das Platform-Team pflegt.
2. **Das Gateway neben den alten Pfaden aufbauen.** Anfangs legt es Session-Logins auf dem Primary an, wo Logins definiert werden, vergibt an sie `human_readonly` und verbindet sie mit `ledgerly_replica_2`, und sonst nichts. Die alten Pfade funktionieren weiter. Ein paar Leute nutzen das Gateway für ihr tägliches Lesen und berichten, was umständlich ist.
3. **Zuerst das Lesen umziehen.** Lesen ist das, was Menschen am meisten tun, und es birgt das geringste Risiko beim Ausprobieren. Sobald die umständlichen Stellen behoben sind, schließen Sie das Replikat für das VPN und löschen die gespeicherten Einträge, die darauf zeigten.
4. **Den Primary zuletzt umziehen.** Der direkte Zugriff auf den Primary ist das, worauf Menschen sich während eines Incidents stützen, daher würde ich ihn erst schließen, wenn die JIT- und Break-Glass-Routen aus Kapitel 8 funktionieren und geübt sind. Ihn früher zu schließen würde den Notfallpfad wegnehmen, bevor der Ersatz bereit ist.
5. **Die alte Bastion und die Reichweite des Admin-Logins entfernen.** Die Bastion des Auftragnehmers wird abgeschaltet. Der gemeinsame Admin-Login wird aus Netzwerken von Menschen nicht mehr akzeptiert. Services behalten ihre eigenen Logins, bis die Arbeit aus Kapitel 4 sie ersetzt.
6. **Die Verbote testen.** Für jeden früheren Pfad probiert jemand ihn aus und bestätigt, dass er fehlschlägt. Ein verweigerter Pfad, den niemand getestet hat, ist eine Annahme.

Die Kosten der Migration sind überwiegend menschlich. Menschen müssen einen neuen Verbindungsablauf lernen, und manche ihrer Skripte werden brechen, weil sie einen dauerhaften Login voraussetzten. Planen Sie ein Fenster, in dem beides funktioniert, und schreiben Sie das Datum auf, an dem der alte Pfad schließt. Ohne Datum bleibt der alte Pfad offen, denn es erscheint immer günstiger, ihn stehen zu lassen.

Abwägungen und Kosten

Ein kleineres Design, das genügen kann. Ein Gateway ist ein neues System im kritischen Pfad, und wenn Sie seinen Rufbereitschaftspfad und seine Deployments nicht besetzen können, ist ein kleineres Design der bessere Anfang. Wenn vier Entwickler über ein privates Netzwerk eine Datenbank erreichen, mit personenbezogenen, kurzlebigen Logins, die etwas ausstellt, das Ihr Team ohnehin betreibt, haben Sie vielleicht schon den größten Teil des Nutzens. Ich würde aufschreiben, was Sie umstimmen würde: ein hinzukommender Auftragnehmer, eine zweite Datenbank, eine Anfrage nach Session-Aufzeichnungen oder der erste Incident, bei dem das Log nicht sagen konnte, wer es getan hat. Das Gateway ist eine Antwort auf diese Bedingungen, kein Meilenstein.

Geschlossen fehlzuschlagen hat einen Preis. Eine Session zu verweigern, wenn das Audit-Log nicht erreichbar ist, hält die Aufzeichnung vollständig und macht den Log-Store zu einer Abhängigkeit jeder Session von Menschen. Manche Teams ziehen es vor, Lese-Sessions mit einem lokalen Puffer zu öffnen und die Einträge später zu senden. Das ist für schreibgeschützte Sessions auf einem Replikat eine vertretbare Wahl, und für Sessions, die schreiben, eine schwächere. Wenn Sie sie treffen, sagen Sie, für welche Sessions sie gilt, und überwachen Sie den Puffer.

Die Anweisungsaufzeichnung ist eine Datenkopie. Der Fingerprint-Eintrag schwärzt Literale, nennt aber weiterhin Tabellen und Spalten, und jedes ungeschwärzte Feld, etwa eine Freitextnotiz oder ein vom Client gelieferter Kommentar, kann einen Wert enthalten. Behandeln Sie das Log mit der Klassifizierung dessen, was ein Eintrag im sensibelsten Fall enthalten könnte, beschränken Sie, wer es liest, und legen Sie eine Aufbewahrungsfrist fest, für die jemand verantwortlich ist.

Latenz und Wiedergabetreue. Ein Gateway im Datenpfad fügt einen Hop hinzu und unterstützt womöglich nicht jedes Client-Feature oder jede Protokollerweiterung. Manche Tools verhalten sich über einen Vermittler seltsam: sehr große Übertragungen, langlebige Cursor und ungewöhnliche Authentifizierungsabläufe sind die üblichen Stellen. Testen Sie die Tools, die Ihre Leute tatsächlich nutzen, bevor Sie versprechen, dass der alte Pfad schließt.

Wann man kein Gateway einsetzt. Ein Prototyp mit einem einzigen Service und ohne Kundendaten braucht keines. Ein System, dessen einziger Zugriff von Menschen über eine verwaltete Konsole läuft, die bereits Identität, kurzlebige Credentials und Logging bietet, braucht womöglich keine zweite Schicht davor. Eine Bastion ist eine vernünftige Antwort, wenn der eigentliche Bedarf ein einzelner Ort zum Schließen der Netzwerktür ist und wenn das geschützte System selbst eine starke, personenbezogene Authentifizierung hat. In beiden Fällen gelten dieselben Fragen: Wer ist die Person, wie lange dauert der Zugriff, und wo ist die Aufzeichnung.

Häufige Fehler

- **Das VPN als Identitätsschicht behandeln.** Es ist verlockend, weil das VPN bereits weiß, wer darin ist, und alle es haben. Es beweist Netzwerkmitgliedschaft, nicht welcher Datenbank-Login verwendet wurde oder was er getan hat. Behalten Sie das VPN als Weg zum Gateway und fügen Sie eine Schicht hinzu, die jede Session an eine Person bindet.
- **Eine Bastion vor die Datenbank stellen und die Arbeit für erledigt halten.** Sie räumt das Netzwerk auf und lässt den Login geteilt und die Anweisungen unsichtbar. Wählen Sie eine Bastion für das, was sie tut, eine einzelne schließbare Netzwerktür, und suchen Sie Identität und Aufzeichnung anderswo.
- **Den alten Pfad während der Migration offen lassen.** Es scheint klug, beide Pfade eine Weile zu behalten, und das ist es auch, aber ohne Schlusstermin überlebt der alte den neuen. Schreiben Sie das Datum auf, kündigen Sie es an und testen Sie danach, dass der alte Pfad fehlschlägt.
- **Eine Session öffnen, deren Ende niemand durchsetzt.** Ein Session-Limit, das nur in der Dokumentation steht, ist ein dauerhafter Grant mit längerem Namen. Beenden Sie Sessions bei `ends_at`, schließen Sie die Verbindungen und löschen Sie nicht nur den Login, und lassen Sie einen Sweep laufen, der Sessions und Logins findet, die nicht existieren sollten.
- **Die Entscheidung über den degradierten Modus überspringen.** Ein Team, das ein Gateway baut, ohne zu entscheiden, was bei dessen Ausfall geschieht, wird unter Druck entscheiden, meist indem es den

alten gemeinsamen Pfad „nur für eine Minute“ wieder öffnet. Entscheiden Sie die Notfallroute vorab, schreiben Sie sie auf und üben Sie sie.

- **Alles aufzeichnen und nichts lesen.** Jede Anweisung festzuhalten fühlt sich nach Sorgfalt an. Ohne einen Verantwortlichen, der Stichproben nimmt, und ein Zugriffs-Review, das hineinsieht, ist das Log eine Datenkopie ohne Zweck. Entscheiden Sie, wer es liest und wonach er sucht.
- **Die Netzwerkliste für präziser halten, als sie ist.** In einem gemeinsamen Cluster-Netzwerk kann die Regel einen Service nicht vom anderen unterscheiden, und ein Reviewer, der annimmt, sie könne es, wird einen Pfad genehmigen, der breiter ist, als er aussieht. Schreiben Sie stattdessen auf, welche Kontrolle die Aufrufer trennt.

Zentrale Erkenntnisse

- Netzwerkreichweite und Datenbankzugriff sind getrennte Fragen, und ein Design, das beide mit derselben Kontrolle beantwortet, kann nicht sagen, welche Person was getan hat.
- Eine Bastion macht den Netzwerkpfad singulär, versteht aber das Datenbankprotokoll nicht und lässt daher für sich allein den Login geteilt und die Anweisungen unsichtbar.
- Ein Access-Gateway authentifiziert eine Person über Single Sign-on, legt einen Session-Login an, der diese Identität trägt, und entfernt ihn, wenn die Session endet; genau das lässt das Datenbank-Log eine einzelne Person nennen.
- Die Netzwerkregel, die nur dem Gateway den Zugang zu den Datenbank-Hosts erlaubt, ist die Kontrolle, die das Gateway zur einzigen Tür macht, und sie sollte eine kurze Liste sein, die ein Zugriffs-Review lesen kann.
- Ein einzelner Einstiegspunkt ist ein Single Point of Failure, daher sollte das Gateway klein sein, in mehr als einer Instanz laufen, dem Platform-Team gehören und mit einer Entscheidung über den degradierten Modus verbunden sein, die vor dem Incident getroffen wurde.
- Das Gateway kann nicht kontrollieren, was eine Person mit Daten tut, die einen Laptop erreicht haben; es kann gespeicherte Credentials ent-

fernen, schreibgeschützte Standardwerte setzen und Exporte sichtbar machen, und der Rest gehört zu Maskierung und Minimierung.

- Ziehen Sie das Lesen zuerst um, schließen Sie den Primary zuletzt, setzen Sie für jeden alten Pfad einen Schlusstermin und testen Sie, dass die verweigerten Pfade verweigert sind.

Das Gateway gibt jeder Person eine Session mit einem Namen darauf und dem Standard Lesen, aber Lesen ist selten die ganze Aufgabe. Das nächste Kapitel betrachtet, was geschieht, wenn jemand für begrenzte Zeit mehr als `human_readonly` braucht: Just-in-Time-Zugriff, Break-Glass-Zugriff und was die Genehmigung für die Menschen kostet, die darauf warten.

Kapitel 8 — Just-in-Time- und Break-Glass-Zugriff

Stellen Sie sich eine Dienstagnacht bei Ledgerly vor, kurz nach drei Uhr morgens. Ein Alarm meldet, dass ein Batch Rechnungen im Status `sending` hängt, seit der `invoice-worker` neu gestartet wurde, und die Erinnerungs-E-Mails für den Morgenlauf gehen nicht hinaus, bis der Batch abgearbeitet ist. Nichts liegt offen und nichts geht verloren; die Kosten des Problems sind eine Verzögerung. Robin, die diensthabende Person im Backend, öffnet über das Access-Gateway eine Routine-Session. Die Session ist schreibgeschützt und läuft auf einem Replikat, und genau das will Robin für die ersten zehn Minuten: ansehen, zählen, verstehen.

Der Blick bestätigt das Problem, und die Korrektur braucht ein `UPDATE` auf einer Spalte von `invoices`. Robin schreibt die Korrektur aus einer einzigen Anweisung auf und stellt dafür einen Antrag. Genehmiger ist die sekundär diensthabende Person, die schläft, und die Alarmierung wurde nicht bestätigt. Robin hat einen Laptop, ein Runbook und eine Entscheidung zu treffen: auf eine Antwort warten oder einen Weg um die Frage herum finden. Im alten Ledgerly hätte es diese Entscheidung nicht gegeben, denn das gemeinsame Admin-Passwort war nur ein Einfügen entfernt. Im neuen Ledgerly muss der Weg drumherum entworfen werden, und der Weg zurück ebenso.

Dieses Kapitel ergänzt die nächsten Schritte zum Zielzustand aus dem durchgehenden Beispiel: Menschen haben keinen Dauerzugriff zum Schreiben, Schreibzugriffe kommen als Just-in-Time-Grants innerhalb der Obergrenze aus Kapitel 3, und es gibt eine Notfallrolle, die selten genutzt und immer geprüft wird. Unter den fünf Teilen des Buches ist es überwiegend der Grant: wer was genehmigt, für welche Anweisung, für wie lange. Das Kapitel beantwortet Robins Frage: Wie entsteht Zugriff, wenn er gebraucht wird, wie endet er mit der Aufgabe, und was geschieht, wenn der normale Pfad zu langsam ist?

Das Problem und warum es schwierig ist

Dauerzugriff ist das Versprechen, dass niemand je wieder fragen muss. Einen Monat lang ist das bequem, danach weiß niemand mehr, was das Versprechen abdeckte. Menschen wechseln das Team und behalten ihre Grants; eine für einen Incident erweiterte Rolle bleibt weit. Wenn ein Zugriffs-Review ansteht (Kapitel 14), steht der Reviewer vor Grants ohne zugeordnete Aufgabe, und die Antwort auf „Wird das noch gebraucht?“ lautet „wahrscheinlich“. Die Kosten eines dauerhaften Grants entstehen an jedem Tag, an dem er nicht genutzt wird, denn er trägt weiterhin den Auswirkungsbereich dessen, was er abdeckt.

Der naheliegende Ersatz ist **Just-in-Time-Zugriff (JIT)**: Zugriff, der für eine bestimmte Aufgabe und eine begrenzte Zeit vergeben wird. Er lässt sich leicht formulieren und an fünf Stellen leicht auf subtile Weise falsch umsetzen.

Reichweite. Die einfache Implementierung vergibt eine Rolle für eine Stunde an einen Login. Ist diese Rolle `human_jit_write`, erhält der Login die gesamte Obergrenze, also Update-Rechte auf sechs Tabellen für eine Aufgabe, die eine Spalte einer einzigen Tabelle brauchte, und der Grant ist ein dauerhafter mit einem Timer daran. Die Reichweite muss in den Grants selbst stecken, Antrag für Antrag.

Ablauf. Auf gängigen Engines laufen Grants nicht von selbst ab. Etwas muss sie zurücknehmen, meist ein geplanter Job, und ein Job kann stehen bleiben, sein Credential verlieren, bei einem Login fehlschlagen und weiterlaufen oder gegen die falsche Liste laufen.

Genehmigung. Dass eine zweite Person jeden Antrag liest, hat menschliche Kosten: Jemand wird geweckt, jemand wartet. Wenn Genehmigungen zu langsam sind, finden Menschen den schnelleren Pfad, nämlich das gemeinsame Passwort, das dieses Buch ablösen will. Antwortet der Genehmiger immer binnen Sekunden, verringert die Genehmigung die Gefährdung nicht mehr.

Der Notfall. Ein Zugriffsschema ohne sanktionierten Weg, schnell zu handeln, bringt einen nicht sanktionierten hervor. Die Frage ist, ob der schnelle Pfad entworfen, protokolliert und geprüft wurde oder improvisiert ist.

Die Operation. Ein Grant benennt eine Tabelle und eine Spalte und sagt nichts darüber, welche Zeilen oder welche Anweisung gemeint sind. Wer „`UPDATE` auf `invoices`, `status`“ liest und genehmigt, hat eine Fähigkeit genehmigt, und alles, was darin läuft, liegt außerhalb der Genehmigung. Damit Genehmigung bedeutet, dass eine zweite Person gesehen hat, was geschehen wird, muss der Antrag die Operation benennen, und der Grant muss aus ihr folgen.

Eine Richtlinie, die besagt „Zugriff muss befristet sein“, sagt dem System nicht, was zu tun ist, wenn der Revoke-Job ausgefallen ist. Der Rest des Kapitels verdrahtet Grants, Datensätze und Timer so, dass es das weiß.

Just-in-Time-Zugriff: Anträge, die eine Operation benennen

Ein JIT-Antrag besteht aus fünf Elementen, und jedes steht wegen eines der obigen Fehler da. Jedes ist mit dem Teil des Modells des Buches gekennzeichnet, dem es dient.

- Die Obergrenze (Grant), `human_jit_write`, legt fest, was ein Antrag höchstens verlangen darf. Sie wird nie an einen Login vergeben (Kapitel 3).
- Ein Schreibantrag (Grant) benennt in `fix_id` die geprüfte Korrektur, für die er gilt. Seine Tripel aus (Privileg, Tabelle, Spalten) werden aus dieser Korrektur innerhalb der Obergrenze abgeleitet und in `access_request_privilege` gespeichert.
- Die Genehmigung (Grant) gilt der Anweisung, und der Grant setzt ihre Grenzen durch: Der Genehmiger liest die Anweisung, das Gateway vergibt genau die abgeleiteten Tripel an den Session-Login des Antragstellers, und die genehmigte Menge von Anweisungen wird als `approved_hash` festgehalten. Der Ablauf widerruft genau diese Tripel, gelesen aus demselben Datensatz.
- Schreibzugriffe in einer JIT-Session (Ausführung) werden nur vom Runner weitergeleitet, der alles zurückweist, was nicht zu `approved_hash` passt.
- Ein unabhängiger Abgleich (Aufzeichnung) vergleicht die Grants, die die Datenbank hält, mit den Anträgen, die sie rechtfertigen, sodass ein

stehen gebliebener Job von etwas anderem bemerkt wird als vom Job selbst.

Ich gehe sie in der Reihenfolge durch, in der ein Antrag ihnen begegnet.

Was ein Antrag enthält

Ein Antrag wird aus der Session gestellt, die ihn nutzen wird, und diese Session läuft auf dem Primary. Eine Session trägt höchstens einen aktiven Antrag; eine Aufgabe, die ihr Zeitfenster überschreitet, wird in einer neuen Session mit einem neuen Antrag fortgesetzt. Routineuntersuchungen bleiben mit `human_readonly` auf dem Replikat (Kapitel 6), und Kapitel 7 legt fest, wer eine an den Primary gebundene Session öffnen darf. Das Gateway weiß, welcher Login zu welcher Person gehört, also tippt niemand einen Principal oder einen Login ein.

So sieht Robins Antrag aus, bevor ihn jemand beantwortet hat:

```
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")
```

Drei Dinge verdienen Beachtung. Der Grund verweist auf etwas Überprüfbares, einen Alarm oder ein Ticket. Der Grant ist schmal, und Robin braucht kein Tripel für `SELECT`, weil die Session bereits `human_readonly` hält. Und Robin hat das Tripel nicht getippt: Es wurde aus `fix-0093` abgeleitet, sodass der Grant und die Anweisung, die der Genehmiger liest, nicht voneinander abweichen können. Ein Antrag, der nur Tripel für `SELECT` enthält, etwa um eine Confidential-Spalte unmaskiert zu lesen, hat keine Korrektur und keinen Hash.

Ein Grant beschränkt Privilegien und Spalten, nicht Zeilen. Das obige `UPDATE` kann `status` bei jeder Rechnung ändern, die die Session sehen kann. Ein JIT-Grant begrenzt nach Spalte und Zeit, was ein Fehler erreichen kann; was ihn zeilenweise begrenzt, ist die Anweisung, die der Genehmiger gelesen hat und der Runner ausführt, zusammen mit der Count-first-Disziplin aus Kapitel 10.

Den Antrag an die Anweisung binden

Die Korrektur ist eine geprüfte Menge von Anweisungen mit ihrem Filter und ihrer erwarteten Zeilenzahl, in der Form, die Kapitel 10 beschreibt, und der Antrag benennt sie über `fix_id` und trägt ihren `approved_hash`. Der Hash stammt aus dem Merge-Datensatz der Korrektur oder aus dem genehmigten Antragsdatensatz und nie aus der Eingabe des Runners, und in einer JIT-Schreibsession leitet das Gateway Schreibzugriffe nur vom Runner weiter. Ein Antrag, der nur Tripel für `SELECT` enthält, hat keine Korrektur, keinen Hash und keinen Runner, und der Genehmiger liest die Spaltenliste. Kapitel 9 formuliert die Regel vollständig und behandelt, wie das Gateway jede Anweisung prüft.

Einen Antrag prüfen

Das Gateway prüft die Form des Antrags, bevor jemand um Genehmigung gebeten wird, und prüft ihn zum Zeitpunkt der Vergabe noch einmal, gemeinsam mit dem Genehmiger.

```
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
```

In den Zeilen zur Selbstgenehmigung wird das Vier-Augen-Prinzip im Code durchgesetzt: Wer den Antrag stellt oder die Korrektur geschrieben hat, kann nicht diejenige Person sein, die antwortet. Ein ausstehender Antrag hat keinen Genehmiger, weshalb die Formprüfung diese Zeilen auslässt; der Genehmiger wird im Moment der Genehmigung eingetragen, und die vollständige Prüfung läuft zum Zeitpunkt der Vergabe.

Ich würde ein zu langes Zeitfenster ablehnen, statt es stillschweigend zu kürzen: Der Antragsteller glaubt dann, eine Stunde zu haben, obwohl es nur zwanzig Minuten sind, und die Überraschung kommt mitten in der Reparatur.

In `withinCeiling` wird die Obergrenze aus Kapitel 3 durchsetzbar. Die Funktion lehnt ein Tripel ab, dessen Tabelle außerhalb der Obergrenze liegt, dessen Privileg auf dieser Tabelle nicht angeboten wird oder dessen Spaltenliste `iban`, `account_holder_name` oder `token_value` enthält. Sie lehnt `INSERT` und `DELETE` auf `bank_accounts` und `payment_tokens` ab, weil eine neue oder entfernte Zeile Restricted-Werte setzt oder vernichtet. Sie lehnt jedes Tripel ohne Spaltenliste ab, mit Ausnahme von `DELETE` und von Tripeln auf Tabellenebene bei einer Engine, die den Fallback mit Secrets-Tabellen verwendet. Und sie lehnt alles ab, was `ledger_entries`, `audit_events` oder die Antragsdatensätze berührt, die keine menschliche Rolle schreibt.

Eine Sache bleibt absichtlich innerhalb der Obergrenze. `UPDATE (customer_id)` auf `bank_accounts` oder `payment_tokens` besteht `withinCeiling`, weil `customer_id` eine Zeile nur ihrem Eigentümer zuordnet und nicht Restricted ist. Die Zeile einem anderen Kunden zuzuweisen ändert aber, wem das Konto oder das Token gehört, weshalb die Ausführungsrichtlinie es auf den Produktpfad leitet (Kapitel 9). Die Checkliste des Genehmigers fängt es von Hand ab.

Wer genehmigt, wie lange, mit welcher Reichweite

Ein JIT-Antrag hat drei Variablen: den Genehmiger, das Zeitfenster und die Reichweite.

Der Genehmiger ist eine Person, die die Tabellen im Antrag versteht und weder Antragsteller noch Autor der Korrektur ist. Bei Ledgerly ist der naheliegende Genehmiger für `invoices` ein weiterer Backend-Entwickler,

meist die sekundär diensthabende Person. Der naheliegende Genehmiger für alles auf `payments` oder `payment_tokens` ist jemand aus dem Team, das `payments-service` besitzt, weil dieses Team weiß, was ein geänderter `status` dort nachgelagert auslöst. Das Platform-Team betreibt die Access-Tools, und ich würde es nicht zum Standard-Genehmiger für Daten machen, die ihm nicht gehören: Es wird zum Engpass und zu einem Genehmiger, der die Änderung nicht beurteilen kann.

```
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
```

Ein leerer Pool schlägt sichtbar fehl und weicht nie auf einen größeren aus. Andernfalls ist die Reihenfolge des Pools die Eskalationsreihenfolge: Die Alarmierung geht an die erste Person, dann an die nächste, innerhalb desselben ausstehenden Antrags, der erst abgelehnt wird, wenn er `expires_at` unbeantwortet erreicht.

Das Zeitfenster sollte so lang sein wie die typische Aufgabe, nicht wie die schlimmste. Angenommen, eine Datenkorrektur dauert üblicherweise zwanzig Minuten, das Formular bietet eine Stunde an, und das Maximum des Gateways beträgt vier Stunden. Einen Grant zu verlängern ist ein neuer Antrag mit einer frischen Genehmigung, keine Änderung des alten, denn eine Änderung würde eine einzelne Genehmigung unbegrenzt dehnen lassen. Wenn eine Reparatur ihr Zeitfenster überschreitet, endet die Session, und jede offene Transaktion wird mit ihr zurückgerollt; eine als einzelne Transaktion geschriebene Reparatur (Kapitel 10) verliert ihre Arbeit also sauber, statt zur Hälfte angewendet zurückzubleiben.

Die Reichweite sind die Tripel und die Anweisung, aus der sie stammen. Der Genehmiger liest beides, und der Genehmigungsbildschirm zeigt die Tripel als die Grants, zu denen sie werden: „UPDATE auf `invoices`

(`status`), bis 04:12“. Ein Antrag, der in einem `SELECT` eine Confidential-Spalte benennt, zeigt diese Spalte unmaskiert, und der Bildschirm sagt das in Worten, nicht nur in der Spaltenliste.

Die Genehmigung erfolgt auf dem eigenen Bildschirm des Gateways, über Single Sign-on mit einer frischen Multi-Faktor-Prüfung beim Klick, nicht als Reaktion unter einer Chat-Nachricht: Die Genehmigung ist die Kontrolle, also muss alles, was sie aufzeichnet, wissen, wer sie erteilt hat.

Was der Genehmiger prüft

Die Entscheidung des Genehmigers ist enger gefasst als „Ist das eine gute Idee?“: die Anweisung genehmigen und den Grant seine Grenzen durchsetzen lassen. Diese Checkliste spiegelt die Checkliste des Reviewers für die Korrektur in Kapitel 10, von der Zugriffsseite aus.

1. Ist der Grund ein überprüfbarer Alarm oder ein Ticket, und ist er noch offen?
2. Tut die Anweisung auf dem Bildschirm, was der Grund sagt? Lesen Sie die Anweisung, nicht die Zusammenfassung des Antragstellers.
3. Begrenzt der Filter die Zeilen, und stimmt die erwartete Anzahl mit dem überein, was der Antragsteller gesehen hat?
4. Decken die abgeleiteten Tripel die Anweisung ab und nicht mehr? Beim Fallback mit Secrets-Tabellen bedeutet „nicht mehr“ „alle Nicht-Restricted-Spalten der Tabelle“, und der Bildschirm sagt das.
5. Ist das Zeitfenster etwa so lang wie diese Aufgabe und unter dem Maximum?
6. Berührt die Anweisung `customer_id` auf `bank_accounts` oder `payment_tokens` oder irgendeine Restricted-Spalte? Dann ist es keine JIT-Datenkorrektur. Schicken Sie sie auf den Produktpfad und lehnen Sie den Antrag ab.
7. Bin ich weder Antragsteller noch Autor, und würde ich diese Genehmigung nächste Woche dem Reviewer ohne Unbehagen erklären?

Wer eine dieser Fragen nicht beantworten kann, fragt den Antragsteller, während der Antrag ausstehend bleibt. Schweigen ist keine Genehmigung, und eine Spaltenliste ebenso wenig.

Die Vergabe

Die Genehmigung ist der Moment, in dem vier Datensätze übereinstimmen müssen: der Antragsdatensatz des Gateways, der Eintrag im Audit-Log, die Grants in der Datenbank und die Session. Keine Transaktion umspannt sie, also kommt es auf die Reihenfolge an und darauf, was jeder Fehler zurücklässt.

```
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)
```

Lesen Sie ihn von den Fehlermodi her. Ist das Audit-Log ausgefallen, wird der Antrag abgelehnt und nichts vergeben: Ein Grant, der sich nicht aufzeichnen lässt, lässt sich nicht prüfen, und ich halte lieber eine Reparatur an, als eine unsichtbare zu schaffen. Stirbt das Gateway nach dem Daten-

satz und dem Log-Eintrag, aber vor den Grants, ist der Antrag genehmigt und ohne Wirkung, was nichts gefährdet. Stirbt es nach zwei von drei Tripeln, hält der Login zwei, der Datensatz hält drei, und der spätere Revoke nimmt alle drei zurück, denn ein Tripel zu widerrufen, das nie vergeben wurde, ist harmlos.

Die letzten Zeilen behandeln ein Race. Das Zeitfenster beginnt mit der Genehmigung nicht neu, sodass die Formprüfung eine Genehmigung ablehnt, die nach `expires_at` eintrifft; übrig bleibt eine Genehmigung, die einen Moment vor Schließen des Zeitfensters durchgeht, und diese Zeilen nehmen deren Grants sofort zurück.

Der Audit-Eintrag trägt neben den Tripeln auch die Korrektur und den Hash, sodass der Genehmiger, die Anweisung und das Zeitfenster in einem Datensatz stehen. Regelwerke für Zahlungsdaten und Finanzberichterstattung fragen in der Regel, wer eine Änderung an Produktionsdaten genehmigt hat, auf welcher Grundlage und für wie lange, und dieser Datensatz beantwortet alle drei Fragen; die Auslegung liegt bei Rechtsabteilung und Compliance.

Den Grant beenden

Ein Grant endet in drei Schichten, und man verlässt sich auf keine von ihnen allein.

Die erste Schicht ist die Session. Das Gateway beendet sie bei `expires_at` und entfernt den Login, ohne dass ein geplanter Job beteiligt ist: Die Uhr, die den Zugriff begrenzt, gehört der Komponente, die ihn geschaffen hat. Eine an den Primary gebundene Session wird mit dem Zwei-Stunden-Limit aus Kapitel 7 geöffnet; bei einem Grant, aus `grantJitAccess` oder `declareBreakGlass`, setzt das Gateway daher das `ends_at` der Session auf das `expires_at` des Antrags zurück, und `endSession` aus Kapitel 7 schließt die Session zu diesem Zeitpunkt; ein Zeitfenster von mehr als zwei Stunden wird also nicht verkürzt.

```
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")
```

Das Beenden selbst übernimmt `endSession`, die einzige Stelle, die Verbindungen schließt, widerruft, was die Anträge der Session vergeben haben, und den Login entfernt; `sweepSessions` ruft dieselbe Funktion auf (Kapitel 7).

Die zweite Schicht ist der Revoke-Job, der denselben Datensatz liest und entfernt, was der Antrag vergeben hat. `revokeRequestAccess` leitet seine Arbeit aus `access_request_privilege` ab: Bei einem JIT-Antrag widerruft die Funktion jedes gespeicherte Tripel, bei einem Break-Glass-Antrag die Rolle `human_breakglass`. Sie liest keine separate Liste, die abweichen könnte, und behandelt einen fehlenden Login als bereits widerrufen.

```
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)
```

Der Revoke läuft weiter, wenn das Audit-Log ausgefallen ist. Die Asymmetrie ist gewollt: Ein Grant ohne Log-Eintrag ist eine Gefährdung, ein Revoke ohne Log-Eintrag dagegen eine Lücke, die sich später füllen lässt.

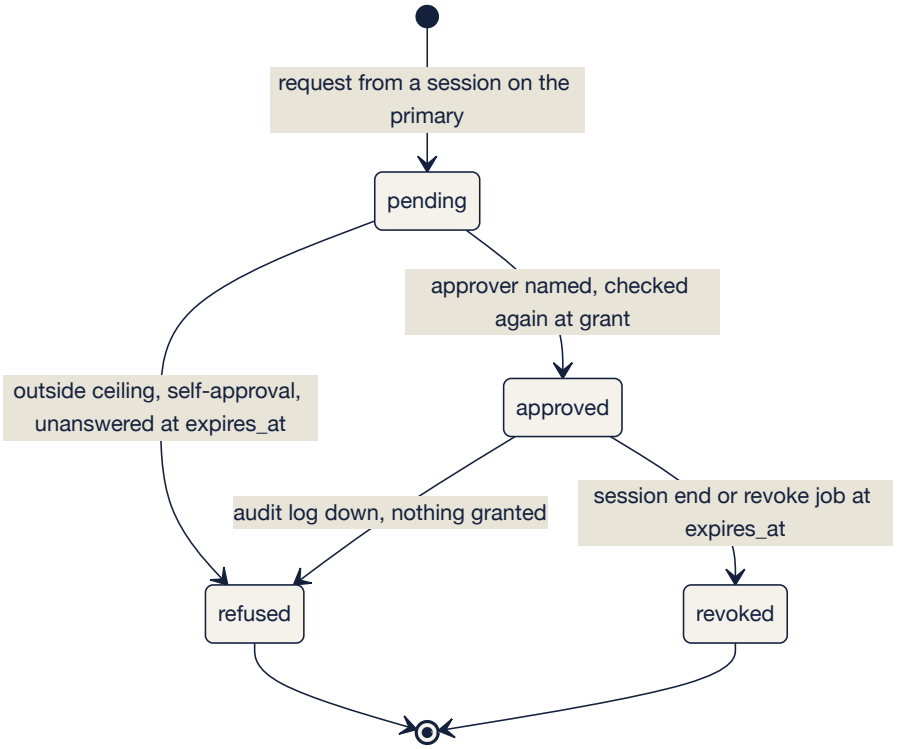
Die dritte Schicht ist der Abgleich, und er ist etwas anderes. Er nimmt keinen Zugriff weg. Er fragt von außen, ob die ersten beiden Schichten ihre Arbeit getan haben:

```
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")
```

Daneben liest die Drift-Prüfung aus Kapitel 3 die Grants, die die Datenbank tatsächlich hält, und vergleicht sie mit den nicht abgelaufenen Tripeln in den Anträgen. Ein direkter Grant ohne aktiven Antrag dahinter, ein Login, der Mitglied der Obergrenze ist, ein Mitglied von `human_breakglass` ohne offenen Break-Glass-Antrag: Jedes wird markiert, selbst wenn der Revoke-Job Erfolg meldet. Die Prüfung abgelaufener Grants beobachtet die Datensätze, auf die der Job hätte reagieren sollen; die Drift-Prüfung beobachtet die Datenbank, die der Job hätte ändern sollen. Ein Job, der „erledigt“ meldet und nichts ändert, wird nur von der zweiten erkannt.

Der Abgleich repariert von sich aus nichts, und ich würde das so belassen: Ein automatischer Revoke unbekannter Grants ist eine Möglichkeit, nachts die Produktion zu beschädigen. Er benachrichtigt das Platform-Team, und ein Mensch entscheidet.

Der Zustand, den ein Antrag durchläuft, ist klein, und sein Zeichnen zeigt, wo jede Schicht wirkt. Sehen Sie sich die Ausgänge von `approved` an: ein Revoke oder eine Ablehnung, wenn das Audit-Log ausgefallen und nichts vergeben ist; aus `revoked` kommt nichts zurück.



Selbstgenehmigung und keine automatische Genehmigung für die Rufbereitschaft

Selbstgenehmigung wird immer abgelehnt. `checkJitRequest` löst `SelfApprovalNotAllowed` aus, und nichts hebelt das aus, auch nicht Seniorität oder Dringlichkeit. Der dringende Fall hat einen anderen Pfad, und der ist die zweite Hälfte dieses Kapitels.

Automatische Genehmigung ist die andere verlockende Abkürzung, weil die Reibung nachts am höchsten ist. Für Lesezugriffe würde ich sie nicht bauen: Jeder Session-Login hält bereits `human_readonly`, und das Einzige, was ein `SELECT` per JIT hinzufügt, ist eine unmaskiert angezeigte Confidential-Spalte, genau das, was ein Genehmiger sich ansehen sollte. Ein Schreibzugriff ändert, worauf die Services als Nächstes reagieren, und eine Maschine kann nicht beurteilen, ob `status` die richtige Spalte ist. Belassen Sie Schreibzugriffe bei einem menschlichen Genehmiger und investieren Sie die Mühe darin, diesen Genehmiger leicht erreichbar zu machen.

Wenn Sie später eine automatische Genehmigung hinzufügen, für einen Lesezugriff, der wirklich etwas Neues mitbringt, muss die Regel eine ausdrückliche Spaltenliste verlangen, und eine leere oder fehlende Liste muss den Antrag sichtbar ablehnen. Beim Fallback mit Secrets-Tabellen hat ein Lesezugriff auf Tabellenebene überhaupt keine Spaltenliste, sodass eine Regel, die „keine benannte Confidential-Spalte“ als akzeptabel behandelt, jede Confidential-Spalte der Tabelle genehmigen würde.

Anträge aus der Automatisierung und die Richtlinie als Code

Ein Job, ein Runbook-Skript oder ein KI-Agent, der schreiben will, ist ein Principal wie jeder andere: Er öffnet seine Session unter seiner eigenen Workload Identity, benennt eine Korrektur und wartet auf einen menschlichen Genehmiger, und der Datensatz nennt sowohl die Automatisierung als auch die Person, die sie ausgelöst hat (Kapitel 9 behandelt, was sie innerhalb einer Session tun darf). Die Genehmigungslogik selbst ist Code. `withinCeiling`, `chooseApproverPool`, `maxJitWindow` und die Formprüfungen liegen im Repository, durchlaufen ein Review und stehen in der Baseline der Drift-Prüfung; eine in einer Konsole von Hand bearbeitete Obergrenze ist das ursprüngliche Problem eine Ebene höher.

Missbrauchsfälle

Drei Fälle werden von den obigen Kontrollen begrenzt und von keiner beseitigt. Kollusion zwischen Antragsteller und Genehmiger wird auf die Spalten, die Anweisung und das Zeitfenster im Datensatz eingegrenzt, und das Paar ist im Audit-Eintrag sichtbar; im Zugriffs-Review (Kapitel 14) fällt ein Paar auf, das einander weit häufiger genehmigt, als der Zufall erklärt. Ein Genehmiger, der per Social Engineering manipuliert wurde („Genehmigen Sie req-0412 jetzt, der Batch blockiert die Gehaltsabrechnung“), trifft auf die Checkliste: die Anweisung auf dem Bildschirm des Gateways lesen, den Alarm im Grund prüfen und nichts beantworten, was über einen anderen Kanal kommt. Ein gekapertes Single-Sign-on-Token kann in einer JIT-Session nicht schreiben, weil das Gateway Schreibzugriffe nur vom Runner weiterleitet und der Runner einen Hash zurückweist, der nie genehmigt wurde. Das Restrisiko liegt bei Lesezugriffen innerhalb der vergebenen Tripel und bei Break-Glass, das keinen Genehmiger hat; die Step-up-Prüfung bei der Erklärung und der Alarm begrenzen es.

Break-Glass-Zugriff

Break-Glass ist Notfallzugriff mit einem definierten Logging-Set und verpflichtendem Review. Der Name stammt vom verglasten Feuerschalter: für alle erreichbar, die ihn brauchen, und sein Einschlagen ist für alle sichtbar.

Im Modell des Buches ist es die Rolle `human_breakglass`, und sie ist die einzige menschliche Rolle, die Restricted-Spalten liest. Sie liest jede Tabelle und schreibt `INSERT` und `UPDATE` auf jeder Tabelle bis auf die beiden Append-only-Tabellen; auf `bank_accounts` und `payment_tokens` schreibt sie nur `UPDATE`, und das nur auf Nicht-Restricted-Spalten. Sie schreibt nie `ledger_entries` oder `audit_events`, weil das keine menschliche Rolle tut. Sie ist kein Weg für Routinekorrekturen und kein Pfad für Datenkorrekturen: Eine Datenkorrektur ist geplant, geprüft und begrenzt (Kapitel 10), und Break-Glass ist nichts davon, und zwar absichtlich.

Ein Break-Glass-Antrag wird selbst erklärt. Er hat keinen Genehmiger. Die Person in der Session sagt, dass ein Notfall vorliegt, nennt einen Grund, und das Gateway markiert den Antrag als genehmigt und vergibt die Rolle an denselben Session-Login bis `expires_at`. Derselbe Revoke-Job und derselbe Session-Timer nehmen sie zurück.

Der Genehmiger ist bei JIT die gesamte Kontrolle, eine Notfallrolle ohne Genehmiger mag daher seltsam wirken. Break-Glass gibt es für die Fälle, in denen der Genehmiger-Pool nicht geantwortet hat, und ein Genehmigungsschritt würde diesen Ausfall wiederholen. Die Kontrolle wandert von vor dem Zugriff an seine Umgebung: ein Alarm bei der Erklärung, das unten beschriebene Logging-Set, ein kurzes festes Zeitfenster und ein Review, das ausnahmslos folgt.

```
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)
```

Drei Entscheidungen in diesem Block brauchen einen Kommentar.

Das Zeitfenster legt das Gateway fest, in der Illustration von Ledgerly auf zwei Stunden. Auf Anfrage gibt es kein längeres Zeitfenster, denn „Ich brauche es noch etwas länger“ macht aus einer Ausnahme eine Gewohnheit. Überschreitet der Notfall das Zeitfenster, erklärt die Person erneut, und die zweite Erklärung alarmiert erneut; das ist die beabsichtigte Reibung.

Die Prüfung des Audit-Logs ist dieselbe wie bei JIT. Ist das Log ausgefallen, wird die Erklärung abgelehnt. Diese Kosten nehme ich in Kauf, weil eine Break-Glass-Session ohne Aufzeichnung das Einzige ist, was ein Review nicht rekonstruieren kann, und das Review macht die Rolle erst tragbar. Was die Rufbereitschaft tut, wenn der normale Pfad und das Log beide ausgefallen sind, gehört in Kapitel 15.

Der Alarm ist eine Benachrichtigung an die Sicherheitskontakte und das Platform-Team, dass jemand den weitesten Zugriff geöffnet hat, den Ledgerly kennt, keine Bitte um Erlaubnis. In einem gesunden System ist er selten genug, um gelesen zu werden; löst er wöchentlich aus, wird er nicht mehr gelesen, und Break-Glass wird für etwas verwendet, das einen entworfenen Pfad verdient.

Das Break-Glass-Logging-Set

„Intensives Logging“ ist kein Standard, solange es nicht definiert ist; für eine Break-Glass-Session besteht das Logging-Set daher aus diesen vier Dingen:

- Jede Anweisung, als Fingerprint, dessen Literale per Redaction entfernt sind, und mit einem Hash der Parameter, mit Beginn, Ende und betroffenen Zeilen. Die Redaction ist hier am wichtigsten: Diese Session liest Restricted-Spalten, und ein Literal im Log würde sie an eine Stelle ohne Schutz auf Spaltenebene kopieren.
- Die Einträge, während sie entstehen. Jede Anweisung wird beim Ausführen an das Audit-Log angehängt, nicht am Ende gesammelt, sodass eine abgebrochene Session trotzdem ihre Aufzeichnung hinterlässt.
- Ein einzeiliger Vermerk zu jedem Schreibzugriff. Das Gateway lehnt einen Schreibzugriff ohne Vermerk ab, was Sekunden kostet und dem Reviewer die Absicht neben der Anweisung liefert.
- Derselbe Datensatz wie bei jeder anderen Session: der Login, die Person, der Antrag und der Grund, damit sich die drei Datensätze aus Kapitel 14 verknüpfen lassen.

Ein Risiko bleibt, und es unterscheidet sich je nach Zustand. Heute, ohne Gateway oder mit einem, das das Credential lediglich vermittelt, ist eine Break-Glass-Session der eine Ort, an dem ein unbegrenztes `UPDATE` auf einer Tabelle, die sie schreiben darf, noch möglich ist. Das Zeitfenster, die Zeilenzahlen, die Einträge und das Review begrenzen und offenbaren den Schaden; sie verhindern ihn nicht. Im Zielzustand führt das Gateway aus Kapitel 9 Schreibsessions mit ausgeschaltetem Auto-Commit und einem Row Cap aus, seine Matrix verweigert in dieser Lane einen unbegrenzten Schreibzugriff, und ein Schreibzugriff ohne Vermerk wird abgelehnt. Möglich bleibt ein Schreibzugriff, der in den Row Cap passt und trotzdem die falschen Zeilen trifft. Der Row Cap lässt sich hier am schwersten bemes-

sen, weil ein Notfall per Definition nicht die geplante Aufgabe ist; solange die Richtlinie keinen setzt, ist der unbegrenzte Schreibzugriff ein akzeptiertes Risiko und der Grund, warum es das Review gibt.

Review im Nachhinein

Jede Nutzung von Break-Glass wird geprüft, und zwar von jemand anderem als der Person, die es genutzt hat. Das Review ist keine Anhörung; es stellt die Fragen eines Design-Reviews: War das ein Notfall, den der normale Pfad nicht bedienen konnte; was hat die Session gelesen oder geschrieben; hat sie Restricted-Spalten gelesen, und war das nötig; und hätte ein JIT-Antrag denselben Bedarf bedienen können, und wenn ja, was hätte diesen Antrag schnell genug gemacht?

Die letzte Frage ist die, die sich auszahlt. Jedes Review bestätigt entweder, dass Break-Glass das richtige Werkzeug war, oder findet eine Lücke im normalen Pfad, die es zum einzigen Werkzeug machte. Die Lücke zu schließen (ein fehlender Genehmiger, ein zu kurzes Zeitfenster, eine Tabelle, die die Obergrenze nicht anbietet) verringert die nächste Nutzung. Das Review braucht außerdem seine eigene Uhr:

```
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)
```

Ein Review, das niemand durchführt, ist eine Kontrolle, die still aufgehört hat zu existieren, daher löst seine Frist einen Alarm aus.

Wenn das Gateway nicht da ist

Menschen erreichen die Datenbank nur über das Access-Gateway (Kapitel 7), und sowohl JIT- als auch Break-Glass-Grants werden vom Gateway an einen Session-Login vergeben, den es selbst angelegt hat. Ein Ausfall des Gateways ist daher auch ein Ausfall von Break-Glass. Das ist eine Abwägung im Entwurf, und Kapitel 7 formuliert dieselbe Entscheidung von der Seite des Gateways aus.

Die Alternative ist eine zweite Tür: ein Notfall-Login außerhalb des Gateways. In der Nacht, in der man sie braucht, ist sie verlockend, und in jeder anderen Nacht gefährlich, denn ein Credential, das ohne das Gateway funktioniert, funktioniert ohne dessen Logging, Identität und Ablauf; es ist das gemeinsame Admin-Passwort unter besserem Namen.

Ich mache das Gateway lieber schwer verlierbar, als seine Abwesenheit überlebbar zu machen, indem ich alles andere aufweiche: mehr als eine Instanz, wenige Abhängigkeiten (das Audit-Log, die Antragsdatensätze), ein geübter Ausfall und ein Health-Signal, das die Rufbereitschaft als Erstes sieht. Sind alle Instanzen gleichzeitig ausgefallen, bleibt kein menschlicher Datenpfad übrig, und ich würde hier keinen erfinden. Das versiegelte Instrument aus Kapitel 3 gibt es für Arbeiten auf Engine-Ebene, es braucht eine Netzwerkroute, die Kapitel 7 für Menschen schließt, und es umgeht die Schutzmaßnahmen für Append-only und Restricted, ist also kein Pfad für den Datenzugriff. Kapitel 15 kommt darauf zurück, was in diesem Fall zu tun ist.

Die Wahl ist nicht kostenlos. Ein Team mit einem fragilen Gateway wird vernünftigerweise einen zweiten Pfad wollen, solange das Gateway reift. Dann schreiben Sie auf, was er ist, wer sein Credential abrufen kann, wie seine Nutzung alarmiert und was ihn ablöst: Ein Pfad mit einer verantwortlichen Person und einem Enddatum ist eine Entscheidung, einer ohne ist ein Überbleibsel.

Wenn die Engine keine Grants auf Spaltenebene kennt

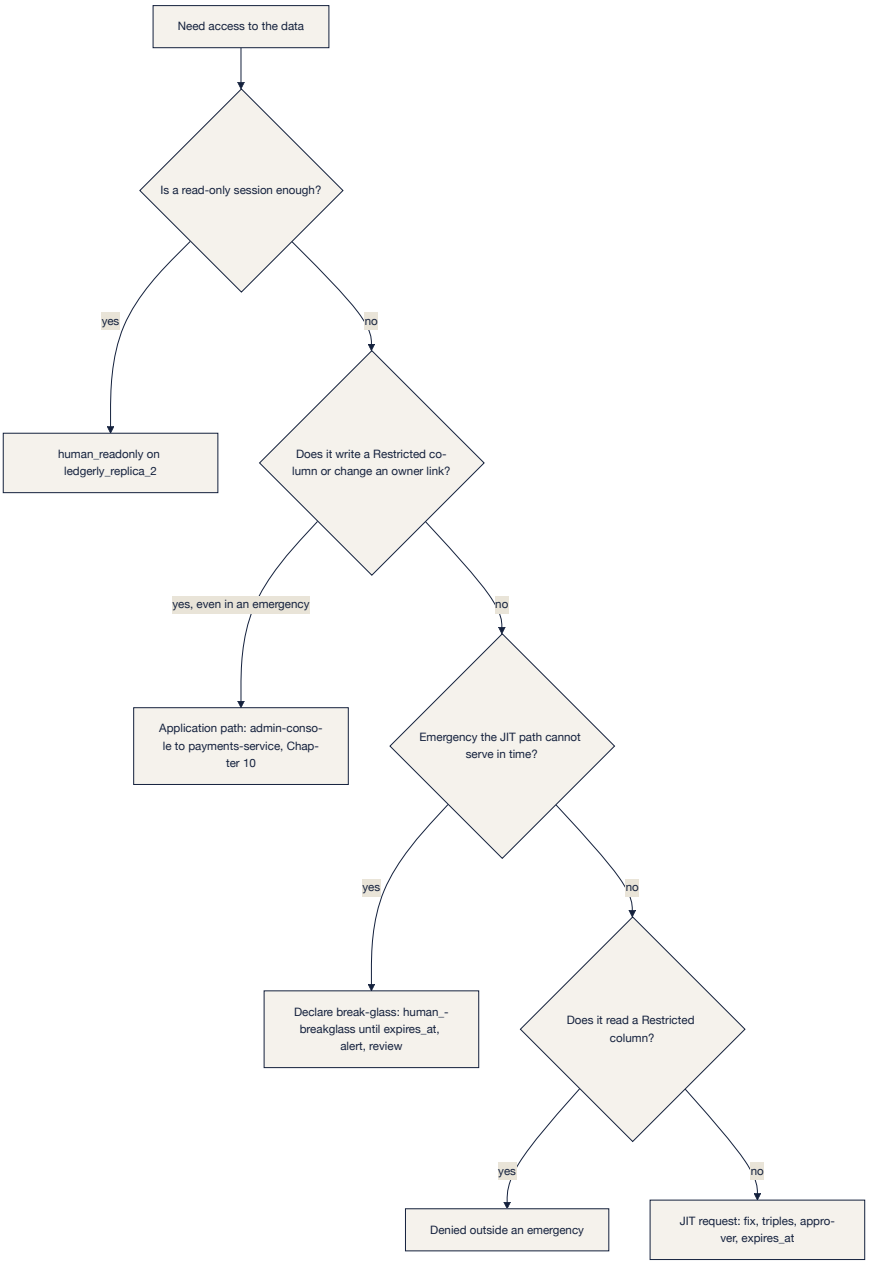
Alles bisher Gesagte setzt voraus, dass die Engine ein Privileg auf benannten Spalten vergeben kann. Kapitel 3 hat den einen Fallback eingeführt, den das Buch für Engines nutzt, die das nicht können: Die Restricted-Spalten liegen in getrennten Tabellen, `bank_account_secrets` und `payment_token_values`, außerhalb der Obergrenze und nur für `app_payments_service` und `human_breakglass` erreichbar. Die Haupttabellen enthalten dann keine Restricted-Spalte, sodass ein Grant auf Tabellenebene für `bank_accounts` oder `payment_tokens` keine erreichen kann. Ein solcher Antrag hat keine Spaltenliste: `columns` ist null, und das Tripel lautet „UPDATE auf `payment_tokens`“.

Den Preis zahlt der Genehmiger, der jede Nicht-Restricted-Spalte der Tabelle genehmigt hat, die Confidential-Spalten eingeschlossen. Wollte der Antragsteller `expires_on` ändern, genehmigt der Genehmiger auch `provider` und `last_four`. Der Genehmigungsbildschirm muss „alle Nicht-Restricted-Spalten dieser Tabelle“ in Worten sagen, und die Anweisung, die der Genehmiger liest, trägt mehr Gewicht. Eine Spaltengruppe, die ein Team häufig zu öffnen erwartet, verdient eine eigene kleine Tabelle.

Der Fallback ändert die Regeln für Restricted-Daten nicht. Eine Person schreibt sie weiterhin nie mit rohem SQL; die Änderung läuft über `admin-console` und `payments-service`, nachdem eine zweite Person genehmigt hat (Kapitel 10).

Den Pfad wählen

Eine Person um drei Uhr morgens muss wissen, welche Tür sie nehmen soll, und die Antwort gehört als kurzer Ablauf ins Runbook. Das folgende Diagramm zeigt seine Form. Achten Sie auf die beiden Enden, die keine menschliche Datenroute bedient: ein Schreibzugriff auf eine Restricted-Spalte, der selbst im Notfall auf den Produktpfad geht, und ein Lesezugriff auf eine solche außerhalb eines Notfalls, der verweigert wird. Der Entwurf sagt das ausdrücklich, statt die Entscheidung jemandem zu überlassen, der unter Druck steht.



Ledgerly um 3 Uhr nachts

Zurück zu Robin. Die hängenden Zeilen wurden in der Routine-Session gezählt, und die Zahl stimmt mit dem Runbook überein. Der Schreibzugriff ist der an `fix-0093` gebundene Antrag, für eine Stunde, mit Verweis auf den offenen Alarm. `chooseApproverPool` wählt die sekundär dienstha-

bende Person und die Teamleitung aus, in dieser Reihenfolge, und die Alarmierung geht an die erste. Das alte Ledgerly hätte hier das gemeinsame Passwort geöffnet. Das neue hat eine Wartezeit, und die Wartezeit ist der Entwurf bei der Arbeit.

Antwortet niemand, stellt das Runbook die Frage, die alles entscheidet: Ist das ein Notfall, den der Genehmiger-Pool nicht rechtzeitig bedienen kann? Hängende Erinnerungen können bis zum Morgen warten, also hinterlässt Robin einen Vermerk am Alarm und hört auf, ein Ausgang, den das alte Passwort nie zuließ.

Angenommen stattdessen, `payments-service` fiele aus, der Zahlungseinzug stünde für alle Kunden still, und die Diagnose bräuchte eine Restricted-Spalte, die kein JIT-Antrag enthalten darf. Das ist ein Fall für Break-Glass. Robin erklärt es mit dem Alarm als Grund; das Gateway vergibt `human_breakglass` für sein festes Zeitfenster; die Sicherheitskontakte und das Platform-Team werden informiert. Am nächsten Morgen liest jemand, der nicht Robin ist, das Log und stellt die Review-Fragen, darunter die, warum die Diagnose eine Restricted-Spalte brauchte. Die Antwort könnte im nächsten Quartal zu einer sichereren Diagnoseansicht führen.

Abwägungen und Kosten

Reibung ist der Preis, und sie muss dort ausgegeben werden, wo sie etwas einbringt. Ein JIT-Antrag kostet den Antragsteller eine Minute zum Schreiben und den Genehmiger eine Minute zum Lesen, im besten Fall, und im schlimmsten Fall eine geweckte Person. Für eine Aufgabe, die zweimal im Quartal anfällt, ist das günstig. Für eine Aufgabe, die zweimal am Tag anfällt, ist es ein Produktfeature im operativen Gewand, und Kapitel 10 beschreibt, wie man es in `admin-console` verlagert.

Die Bindung an eine Korrektur kostet Vorbereitung. Die Korrektur muss aufgeschrieben sein, bevor der Antrag gestellt werden kann, was langsamer ist als ein Grant allein. Das ist der Preis dafür, zu genehmigen, was laufen wird, statt dessen, was laufen könnte.

Genehmigungsmüdigkeit ist ein Versagen des Entwurfs, nicht des Genehmigers. Eine Genehmigung, die jedes Mal innerhalb von Sekunden von derselben Person kommt, ist Theater: Entweder sind die Anträge trivial, und `human_readonly` sollte sie abdecken, oder der Genehmiger hat

aufgehört zu lesen. Das Mittel sind kleinere Anträge, ein Bildschirm, der zeigt, was die Grants sein werden, Genehmiger, die die Tabellen kennen, und ein Pool, der groß genug ist, dass niemand der Standard ist. Zwei Signale gehören ins Zugriffs-Review: wie oft Anträge abgelehnt oder zurückgeschickt werden und wie oft derselbe Genehmiger antwortet. Eine Ablehnungsquote von null deutet ebenso wahrscheinlich darauf hin, dass niemand hinschaut, wie darauf, dass die Anträge gut sind.

Die Rufbereitschaft trägt die Kosten zuerst. Der Genehmiger wird geweckt, der Antragsteller wartet, und das Team lernt ein Runbook, das es selten nutzt. Machen Sie den Genehmiger-Pool zu einer Rotation, veröffentlichen Sie ihn und stellen Sie sicher, dass die Personen darin den Kontext haben, einen Antrag zu `invoices` oder `payments` zu lesen.

Das Gateway ist eine Abhängigkeit jedes Schreibzugriffs. Der Revoke-Job, das Audit-Log und die Antragsdatensätze sind allesamt Abhängigkeiten eines Grants; fällt eines aus, bleiben Grants aus, was die Gefährdung begrenzt und die Incident Response verlangsamt. Kapitel 7 beschreibt, wie sich die Wahrscheinlichkeit senken lässt.

Der Abgleich erzeugt Rauschen. Die Drift-Prüfung und die Prüfung abgelaufener Grants werden legitime Fälle markieren, etwa einen Revoke, der eine Minute zu spät läuft. Stimmen Sie die Schonfrist so ab, dass eine Markierung etwas bedeutet; eine stummgeschaltete Prüfung ist schlimmer als keine, weil das Team glaubt, abgesichert zu sein.

Break-Glass hat soziale Kosten. Wer es erklärt, teilt den Sicherheitskontakten mit, dass der normale Pfad versagt hat. Fühlt sich das Review wie ein Prozess an, zögern Menschen in echten Notfällen; ist es ein Design-Review, tun sie es nicht. Ein schuldzuweisungsfreies Review hält die Rolle nutzbar.

Wann man dies nicht einsetzt. Ein Team aus vier Entwicklern mit einer risikoarmen Datenbank und ohne Restricted-Daten braucht am ersten Tag keine Genehmiger-Pools und keinen Abgleich-Job. Ein manueller Prozess mit einem Grund, einer Zeitgrenze, einer bestätigenden Person und einem Datensatz kann eine Weile genügen, wenn die Grants entfernt werden, sobald die Zeit abläuft, und jemand nachprüft. Bauen Sie das Pattern in der Reihenfolge seiner Ausfälle auf: zuerst den Ablauf, dann den Datensatz, dann den Genehmiger, dann den Alarm.

Häufige Fehler

- **Die Obergrenzen-Rolle „für diese eine Aufgabe“ an einen Login vergeben.** Dann hält jede JIT-Session die gesamte Obergrenze, und die Reichweite des Antrags ist Dekoration. Vergeben Sie die Tripel an den Session-Login und belassen Sie `human_jit_write` als Obergrenze, der kein Login angehört; die Drift-Prüfung markiert ein Mitglied.
- **Die Tripel genehmigen und nicht die Anweisung.** Eine Spaltenliste ist kurz und leicht zu lesen, und sie zu genehmigen heißt, eine Fähigkeit zu genehmigen. Binden Sie den Schreibantrag an eine geprüfte Korrektur, leiten Sie die Tripel daraus ab und nehmen Sie `approved_hash` aus dem Datensatz der Korrektur.
- **Von einer separaten Liste widerrufen.** Eine fest verdrahtete Tabellenliste verpasst irgendwann eine Tabelle, die der Antrag vergeben hat. Leiten Sie den Revoke aus `access_request_privilege` ab, dem Datensatz, der den Grant angetrieben hat.
- **Dem Revoke-Job allein vertrauen.** Einen Job, der ein Jahr lang jede Nacht gelaufen ist, hört man leicht auf zu beobachten. Geben Sie der Session ihre eigene Endzeit und führen Sie eine unabhängige Prüfung aus, die Grants mit Anträgen vergleicht.
- **Den Audit-Eintrag nach dem Grant schreiben.** Ist das Log ausgefallen, existiert ein Grant, den niemand sehen kann. Schreiben Sie den Datensatz, dann den Log-Eintrag, dann die Grants, und lassen Sie einen fehlgeschlagenen Log-Eintrag den Grant stoppen.
- **Anträge der Rufbereitschaft automatisch genehmigen.** Lesezugriffe brauchen keinen Antrag, und ein Schreibzugriff braucht eine zweite Person. Verkürzen Sie die Wartezeit mit einem guten Genehmiger-Pool.
- **Break-Glass so lang machen, dass es bequem wird.** Ein Zeitfenster, das innerhalb eines Arbeitstages nie abläuft, wird zu einer Arbeitsrolle. Legen Sie das Maximum im Gateway fest und lassen Sie eine zweite Erklärung erneut alarmieren.
- **Das Logging-Set undefiniert lassen.** Ein Reviewer kann eine Session nicht beurteilen, deren Aufzeichnung nie festgelegt wurde. Benennen Sie das Set vor der ersten Nutzung.
- **Das Review überspringen, weil nichts schiefgegangen ist.** Im Review wird der normale Pfad repariert. Geben Sie dem Review eine

Frist und lösen Sie einen Alarm aus, wenn sie verstreicht.

- **Eine zweite Tür für den schlechten Tag des Gateways bauen.** Ein Notfall-Login außerhalb des Gateways umgeht dessen Identität, Logging und Ablauf. Investieren Sie zuerst in Verfügbarkeit; ist ein zweiter Pfad unvermeidlich, geben Sie ihm eine verantwortliche Person, einen Alarm und ein Enddatum.

Zentrale Erkenntnisse

- Dauerzugriff kostet an den Tagen am meisten, an denen er nicht genutzt wird; die nützliche Frage lautet daher, wer jetzt Zugriff braucht, für welche Aufgabe und woran das System erkennt, dass die Aufgabe beendet ist.
- `human_jit_write` ist eine Obergrenze und wird nie an einen Login vergeben; ein Schreibantrag benennt seine geprüfte Korrektur (`fix_id`), die Tripel werden aus der Korrektur abgeleitet, und der Genehmiger genehmigt die Anweisung, deren Hash (`approved_hash`) der Runner treffen muss.
- Der Widerruf wird aus demselben Datensatz `access_request_privilege` abgeleitet, der den Grant angetrieben hat, durch die eigene Endzeit der Session erzwungen, von einem Revoke-Job gestützt und von einem unabhängigen Abgleich geprüft, der keinem von beiden traut.
- Schreibzugriffe brauchen das Vier-Augen-Prinzip: einen Genehmiger, der weder Antragsteller noch Autor der Korrektur ist und auf dem Bildschirm des Gateways nach einer frischen Multi-Faktor-Prüfung antwortet. Es gibt keine automatische Genehmigung für die Rufbereitschaft.
- Break-Glass ist eine selbst erklärte Session mit `human_breakglass` mit einem festen Maximalzeitfenster, einem definierten Logging-Set, einem Alarm bei der Erklärung und einem verpflichtenden Review durch jemand anderen als den Nutzer. Es ist weder ein Routineweg noch ein Pfad für Datenkorrekturen, und sein einziges akzeptiertes Risiko ist heute ein unbegrenztes `UPDATE` und im Zielzustand, sobald das Gateway aus Kapitel 9 einen Cap durchsetzt, falsche Zeilen innerhalb des Caps.

- Ein Entwurf ohne sanktionierten schnellen Pfad bringt einen nicht sanktionierten hervor, daher wird der Notfallweg entworfen, protokolliert und geübt, und seine Abhängigkeit vom Gateway ist ein Kostenpunkt, den Sie benennen.
- Wo die Engine nicht auf Spalten vergeben kann, hält der Fallback mit Secrets-Tabellen Restricted-Spalten aus jedem Antrag heraus, und der Genehmiger genehmigt ganze Tabellen aus Nicht-Restricted-Spalten.

Ein Grant, auch ein gebundener, sagt, was eine Session berühren darf und wie lange, nicht aber, wie sorgfältig die Anweisung dahinter geprüft wird. Kapitel 9 wendet sich dem zu: was das Gateway mit einer Anweisung tut, bevor sie läuft. Kapitel 10 wendet sich dann der Arbeit selbst zu: wie man eine Datenkorrektur ausführt, sodass die richtigen Zeilen sich ändern, die falschen nicht und eine zweite Person sehen kann, warum.

Kapitel 9 — Vermittelte Ausführung: Abfrageanalyse und gefährliche Operationen

Das Support-Team von Ledgerly hat einen Assistenten in seinem Chat-Tool. Sam bittet ihn, einen Batch doppelter Rechnungen aus einem Billing-Retry zu stornieren, und der Assistent entwirft unter seiner eigenen Identität eine Korrektur: ein `UPDATE` auf `invoices` mit einer Klausel `WHERE` auf das Batch-Label des Retry. Jordan aus dem Data-Team prüft die Anweisung und genehmigt sie. Die Anweisung ist gültiges SQL, der Grant deckt sie ab, der Genehmiger hat sie gelesen, und das Label war korrekt, als Jordan es prüfte.

Die Anweisung trifft trotzdem mehr als die zwölf Zeilen, die die Korrektur benannte. Ein zweiter Job hat das Batch-Label zwei Tage zuvor wiederverwendet, und nichts im Text der Anweisung sagt das. Niemand war nachlässig: Sam hat die Aufgabe korrekt beschrieben, der Assistent hat geschrieben, worum er gebeten wurde, und Jordan hat geprüft, was ihm gezeigt wurde. Es fehlte ein Schritt zwischen der Genehmigung und dem Commit, ein Ort, an dem etwas darauf schaut, was die Anweisung mit den Daten tun wird, und nicht nur darauf, was sie sagt.

Die Kapitel 3 bis 8 haben entschieden, wer einen Grant halten darf, über welchen Pfad und für wie lange. Dieses Kapitel ergänzt den Teil Ausführung des Modells aus fünf Teilen: den Schritt, in dem das Gateway jede Anweisung behandelt, statt eine Session auszuhändigen und dem zu vertrauen, der sie hält. Die Frage, die es beantwortet, lautet: Was sollte zwischen einer genehmigten Anweisung und den Daten stehen, wenn die Genehmigung allein nicht sagen kann, was die Anweisung tun wird?

Das Problem und warum es schwierig ist

Ein Session Handout gibt einer Person einen Login und lässt deren Client direkt mit der Datenbank sprechen. Alles Weitere ist Sache des Clients: was getippt wird, wie viele Zeilen es berührt, ob es sofort mit einem Commit abgeschlossen wird. Der Grant begrenzt, welche Tabellen und Spalten erreichbar sind, und nichts begrenzt, was eine einzelne Anweisung innerhalb dieser Reichweite tut. Im Ausgangszustand schickt ein Desktop-Client ab, was getippt wird, und führt den Commit sofort aus.

Die vermittelte Ausführung kehrt diese Anordnung um. Die Anweisung geht an eine Komponente, die nicht der Client der Person ist, und diese Komponente entscheidet, ob sie läuft, mit welchen Limits und wann sie mit einem Commit abgeschlossen wird. Das Gateway aus Kapitel 7 liegt bereits im Datenpfad und ist deshalb der naheliegende Ort dafür. Die Person schreibt die Anweisung weiterhin selbst, oder eine Automatisierung tut es; geändert wird, dass keine Anweisung die Daten erreicht, ohne die Schleife im nächsten Abschnitt zu durchlaufen.

Das ist ein Spektrum und kein Schalter. Die günstigste Form ist ein Session Handout mit einigen Einstellungen, die das Gateway durchsetzt, etwa einem Statement-Timeout und einem Row Cap. Die nächste Form prüft jede Anweisung, bevor sie weitergeleitet wird. Die am stärksten vermittelte Form genehmigt jede Anweisung einzeln und führt sie von einem Worker aus, sodass die Person für den Schreibzugriff nie eine Datenbank-Session hält. Manche Tools genehmigen jede Abfrage statt eines Zeitfensters. Zum Zeitpunkt des Schreibens lässt beispielsweise QueryProxy eine zweite Person die Anweisung genehmigen und einen Worker sie ausführen, sodass der Entwickler nie das Credential der Datenbank hält. Das tauscht die Geschwindigkeit einer interaktiven Session gegen eine Genehmigung, die an die genaue Anweisung gebunden ist, und andere Tools zur Abfragegenehmigung arbeiten ähnlich.

Vermittlung ist aus drei Gründen schwierig. Der erste: SQL ist ausdrucksstark; dieselbe Wirkung lässt sich auf viele Arten schreiben, und ob eine Anweisung „gefährlich“ ist, hängt oft von Daten ab, die der Prüfer nicht gelesen hat. Der zweite: Jede Prüfung kostet etwas, nämlich Latenz, falsche Ablehnungen und Menschen, die lernen, sie zu umgehen. Der dritte: Ein Prüfer, dem zu sehr vertraut wird, wird zu dem, was niemand mehr prüft. Die Position dieses Kapitels ist, dass die Anweisungsanalyse Fehler

verringert und nicht die Autorisierungsgrenze ist. Die Grants sind die Grenze. Die Aufgabe des Gateways ist es, die häufige falsche Anweisung schwer versehentlich ausführbar zu machen und dafür zu sorgen, dass jeder Schreibzugriff eine Schranke trägt, die etwas anderes als der Schreibende durchsetzt.

Die Schleife des Gateways für jede Anweisung

Jede Anweisung, die eine Session sendet, ob von einem Menschen oder nicht, durchläuft dieselbe Schleife. Die Reihenfolge zählt mehr als jeder einzelne Schritt, deshalb steht sie hier als Pseudocode, bevor die Teile erklärt werden.

```
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
```

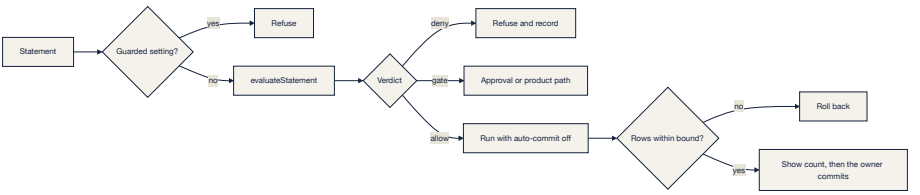
Guarded Settings. Das Gateway öffnet jede Session mit angewandten Limits: dem Statement-Timeout, einem schreibgeschützten Standard, der Rolle und dem Verbindungslimit je nach Art der Session (Kapitel 5). Ein Client kann ein `SET` senden, das jedes davon aufhebt; deshalb führt das Gateway eine Liste von Guarded Settings und lehnt eine Anweisung ab, die eines davon ändert. Das Wort „nicht überschreibbar“ verdient man nur dort, wo ein Mechanismus es durchsetzt. Ein als Rollenstandard gesetzter Timeout kann von der Session aufgehoben werden, sofern das Gateway das `SET` nicht ablehnt oder die Plattform ihn nicht festschreibt; ein Limit,

das keines von beidem hat, ist ein Standardwert und keine Garantie. Die Kosten: Manche Clients senden solche Einstellungen beim Verbinden und funktionieren nicht mehr, bis jemand sie auflistet.

Der Row Cap. Bei Lesezugriffen zählt das Gateway die Zeilen, während es sie streamt, schließt den Cursor am Row Cap, teilt es der Person mit und zeichnet das Ereignis auf. Der Row Cap besagt nicht, dass eine Person über einen Tag hinweg nur so viele Zeilen sehen darf, und eine entschlossene Person kann eine Tabelle in Scheiben durchblättern. Er besagt, dass eine einzelne fehlerhafte Abfrage nicht eine ganze Tabelle auf einen Laptop zieht. Große legitime Auszüge gehören auf den Export-Pfad aus Kapitel 13 und die maskierte Analytics-Kopie aus Kapitel 12, wo die Stufe entscheidet, wer genehmigt.

Auto-Commit aus bei Schreibzugriffen. Eine Schreib-Session führt jeden Schreibzugriff innerhalb einer expliziten Transaktion aus. Bei einer Person oder einer Migration öffnet das Gateway sie; bei einer Datenkorrektur öffnet sie der Runner aus Kapitel 10, und das Gateway leitet jede Anweisung der Korrektur in diese eine Transaktion weiter. In beiden Fällen zählt das Gateway die betroffenen Zeilen über alle Anweisungen hinweg und vergleicht sie mit der Schranke, während die Entscheidung für das `COMMIT` beim Eigentümer der Transaktion bleibt: Ein Gateway, das nach jeder Anweisung einer Korrektur für sich einen Commit ausführte, würde die einzelne Transaktion des Runners zerbrechen und seinen Dry Run zu einer echten Änderung machen. Übersteigt die Zählung die Schranke, lehnt das Gateway die Anweisung ab und markiert die Transaktion als fehlgeschlagen, sodass der Eigentümer nur noch zurückrollen kann. Das ist die günstige Form der Umkehrbarkeit aus Kapitel 1: Ein Update, das eine Zeile treffen sollte und vierhundert meldet, wird gestoppt, solange das Rückgängigmachen noch kostenlos ist. Bei einer interaktiven Session heißt „angezeigt“, dass eine Person die Zahl sieht und bestätigt. Beim Runner und bei der Automatisierung heißt „angezeigt“, dass das Gateway die Zahl mit der Schranke vergleicht, die die Genehmigung festgehalten hat, und die Transaktion bei jeder Überschreitung fehlschlagen lässt. Die Kosten sind der zusätzliche Schritt und die Sperren der Transaktion, die gehalten bleiben, während die Person entscheidet; eine Bestätigungsaufforderung, die niemand beantwortet, hält eine Zeilensperre, daher hat die Aufforderung einen Timeout, der zurückrollt.

Zuerst die Aufzeichnung. Das Gateway schreibt die Absicht, bevor die Anweisung läuft, und das Ergebnis danach. Keine Transaktion umspannt das Log des Gateways und die Datenbank, also trägt die Reihenfolge die Garantie. Kann die Absicht nicht geschrieben werden, läuft die Anweisung nicht. Kann das Ergebnis nach einem Commit nicht geschrieben werden, bleibt der Commit bestehen, der Eintrag kommt in eine Warteschlange, und das Platform-Team wird alarmiert; ein Log, das nur funktioniert, wenn nichts schiefgeht, verdient den Namen nicht. Kapitel 14 behandelt, wo die Aufzeichnung liegt.



Eine Anweisung klassifizieren

Das Gateway entscheidet in `evaluateStatement`, was mit einer Anweisung geschieht. Es klassifiziert die Anweisung danach, was sie tut und wo, schlägt die Matrix im übernächsten Abschnitt nach und findet bei einem Schreibzugriff, den die Matrix mit einer Schranke erlaubt, diese Schranke und vergleicht die Schätzung des Planers damit. Jede Ablehnung und jedes Gate stammt aus der Matrix; die Funktion hat außer denen des Parsers keine eigenen vorzeitigen Rückgaben.

```
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)
```

Drei Entscheidungen verdienen eine Verteidigung. Erstens wird eine Anweisung abgelehnt, die der Parser nicht lesen kann. Würde das Gateway ausführen, was es nicht analysieren konnte, würde die Analyse nur für die Anweisungen gelten, die nie ein Problem waren. Ein Client, der Batches von Anweisungen in einer Zeichenkette sendet, bricht unter dieser Regel, und das ist ein Kostenpunkt, den die Migration benennen muss; die Antwort lautet, sie einzeln zu senden.

Zweitens betrachtet die Klassifizierung die Wirkung und nicht den Namen der Person oder des Tools. Ein `DELETE` in einer Migration und ein `DELETE` in einer JIT-Session sind dieselbe Klasse, und die Matrix entscheidet, was jede Lane damit tun darf. Ein Klassifikator mit einer Tabelle statt eines eigenen Regelwerks je Pfad hält die Richtlinie klein genug zum Prüfen.

Drittens wird die Schätzung des Planers mit der erwarteten Anzahl verglichen, aber nur als frühe Ablehnung. Der Planer schätzt aus Statistiken, und Statistiken veralten; eine Schätzung kann ebenso leicht zu niedrig wie

zu hoch sein. Die Schätzung lässt das Gateway eine offensichtlich falsche Anweisung ablehnen, bevor sie Sperren nimmt. Die Kontrolle ist die echte Zählung, die innerhalb der Transaktion vor dem `COMMIT` verglichen wird. Die Toleranz ist ein Parameter, den das Platform-Team setzt und festhält, und ich würde eng beginnen und sie erst lockern, wenn falsche Ablehnungen gemessen und nicht vermutet werden.

Was die statische Analyse nicht sieht

Der Text einer Anweisung sagt wenig über ihre Wirkung auf Zeilen. Ein Parser kann sehen, dass ein `WHERE` fehlt, dass ein Prädikat immer wahr ist, dass eine Anweisung eine Tabelle berührt, die sie nicht berühren sollte, oder dass sie eine Funktion aufruft. Er kann nicht sehen, wie viele Zeilen ein Prädikat heute trifft, und das ist das Problem vom Anfang: ein gültiger Filter auf einem Label, das jetzt mehr Zeilen trifft, als jemand erwartet hat.

Mehrere weitere Fälle liegen außerhalb dessen, was Analyse entscheiden kann. Ein Filter kann gültig und trotzdem breit sein und den Großteil einer Tabelle treffen, wegen der Daten und nicht wegen der Syntax. Ein Join kann Zeilen vervielfachen. Eine View oder eine Funktion kann einen Schreibzugriff oder einen breiteren Lesezugriff hinter einem harmlos wirkenden Namen verbergen, und das Gateway kann nicht in einen Funktionskörper schauen, den die Rolle aufrufen darf. Ein ausgelöster Nebeneffekt geschieht nach der Anweisung und ist in ihrem Text unsichtbar. Und eine Anweisung kann der Form nach genau richtig und der Bedeutung nach falsch sein: dem Filter nach die richtigen Zeilen, der Absicht nach die falschen, was kein Tool beurteilen kann.

Open-Source-Tools wie QueryProxy prüfen Anweisungen bei der Einreichung, weisen ein `UPDATE` oder `DELETE` zurück, dessen `WHERE` fehlt oder immer wahr ist, und begrenzen unbegrenzte `SELECT`; zum Zeitpunkt des Schreibens beschreibt seine eigene Dokumentation diesen Parser als Verteidigungsschicht und nicht als Autorisierungsgrenze, was genau der Punkt dieses Abschnitts ist, und andere Gateways und Genehmigungs-Tools gehen ähnlich vor.

Daraus folgt eine Arbeitsteilung. Die Analyse fängt die häufigen Fehler günstig und früh ab. Die Grants (Kapitel 3) legen fest, was überhaupt erreichbar ist, sodass eine übersehene Anweisung weiterhin durch die Rolle begrenzt ist. Die Zählprüfung in der Transaktion fängt den Fall ab, in dem die Daten dem Autor widersprachen. Die Genehmigung, bei der eine zweite Person die Anweisung und die erwartete Anzahl liest, fängt den Fall ab, in dem die Form stimmte und die Absicht nicht, zumindest manchmal. Keine davon ist vollständig, und der Entwurf stützt sich auf ihre Überlappung und erhebt keine von ihnen zur Grenze.

Dieselbe Beschränkung prägt die Export-Stufe aus Kapitel 13. Das Gateway berechnet die Stufe eines Exports aus den Quellspalten, die die Abfrage liest, nicht aus den Namen der Ergebnisspalten, und eine Quelle, die es nicht auflösen kann, zählt als höchste exportierbare Stufe; wo auch das nicht entschieden werden kann, wird der Export abgelehnt. Das ist dieselbe Fail-closed-Regel wie beim Parser: Was die Analyse nicht lesen kann, winkt sie nicht durch.

Die Matrix aus deny, gate und allow

Jede Anweisungsklasse erhält in jeder Lane eine von drei Antworten. **Deny** heißt, das Gateway lehnt ab und zeichnet die Ablehnung auf. **Gate** heißt, die Anweisung läuft nicht wie gesendet: Sie braucht eine Genehmigung, die sie noch nicht hat, oder gehört auf einen anderen Weg. **Allow** heißt, sie läuft unter den Limits der obigen Schleife; wo eine Zelle „Schranke“ sagt, ist der Schreibzugriff nur innerhalb einer Schranke erlaubt, die aus der Quelle stammt, die die Zelle nennt, und die echte Zählung wird vor dem `COMMIT` damit verglichen. Die Spalten sind die Lanes, die das Buch beschrieben hat, und die Zeilen die Anweisungsklassen, die der Klassifikator erzeugt.

Anweisungsklasse	Lese-Session für Menschen	JIT-Schreibzugriff	Break-Glass	Runner für Datenkorrekturen	Migration
Lesen, innerhalb des Row Cap	Allow	Allow	Allow	Allow	Allow

Anweisungsklasse	Lese-Session für Menschen	JIT-Schreibzugriff	Break-Glass	Runner für Datenkorrekturen	Migration
Begrenzter Schreibzugriff, innerhalb der genehmigten Anzahl	Deny	Allow, an die Korrektur gebunden (nur Runner)	Allow, an einen Row Cap gebunden (Vermerk erforderlich)	Allow, an den Approved Hash und die Anzahl gebunden	Allow, an die Backfill-Regel gebunden
Unbegrenzter Schreibzugriff	Deny	Deny	Deny	Deny	Gate: benannter Genehmiger
Änderung der Eigentümerverknüpfung	Deny	Gate: Produktpfad	Gate: Produktpfad	Gate: Produktpfad	Gate: benannter Genehmiger
Schreibzugriff auf eine Restricted-Spalte	Deny	Gate: Produktpfad	Gate: Produktpfad	Gate: Produktpfad	Gate: benannter Genehmiger
Export	Gate: nach Stufe	Deny	Deny	Deny	Gate: benannter Genehmiger
Schemaänderung (DDL)	Deny	Deny	Deny	Deny	Allow: Lane mit Review
Änderung von Privilegien	Deny	Deny	Deny	Deny	Gate: Plattform-Review
Schreibzugriff auf `ledger_entries` oder `audit_events`	Deny	Deny	Deny	Deny	Gate: Plattform-Review

Zeile für Zeile gelesen, zeigt sich der Entwurf. Die Lese-Session für Menschen hat keinen Schreibpfad, also ist dort jede Schreibklasse verweigert, und das einzige Gate ist die Export-Stufe. Eine JIT-Schreib-Session leitet Schreibzugriffe nur vom Runner weiter, sodass ihre Zelle für den begrenz-

ten Schreibzugriff ein Allow mit Bindung ist und keine offene Tür. Break-Glass ist die eine Lane, in der eine Person direkt schreibt, weshalb ihr begrenzter Schreibzugriff nur unter einem Row Cap und mit Vermerk erlaubt ist und weshalb die Größe dieses Cap die schwierigste Bemessungsentscheidung im Buch ist: Ein Notfall ist nicht die geplante Aufgabe, also lässt sich kein Cap aus der Aufgabe wählen. Der Cap sollte danach festgelegt werden, was die Break-Glass-Rolle plausibel zu reparieren hätte, und nach jeder Nutzung geprüft werden. Ein unbegrenzter Schreibzugriff wird dort verweigert. Übrig bleibt ein Schreibzugriff, der innerhalb des Cap bleibt und die falschen Zeilen trifft, ein akzeptiertes Risiko, das das Review aufdecken soll, wie Kapitel 8 festhält; Kapitel 8 sagt auch, was heute gilt, bevor ein Gateway den Cap durchsetzt.

Die Änderung der Eigentümerverknüpfung verdient eine eigene Zeile. Ein `UPDATE` von `customer_id` auf `bank_accounts` oder `payment_tokens` passiert die JIT-Obergrenze, weil die Spalte eine Zeile mit ihrem Eigentümer verknüpft und nicht Restricted ist. Die Zeile einem anderen Kunden zuzuweisen ändert dennoch, wessen Auszahlungskonto oder Zahlungs-Token es ist. Der Klassifikator benennt sie deshalb als eigene Klasse, und jede Lane außer einer Migration antwortet mit einem Gate, das sie auf den Produktpfad leitet: `admin-console` fragt `payments-service`, eine zweite Person genehmigt, und die Änderung wird in `audit_events` aufgezeichnet. Eine Migration in der normalen Lane lässt sie bei einem benannten Genehmiger durch ein Gate laufen, wie Kapitel 11 es bei jeder markierten Anweisung tut, und die beschleunigte Lane lehnt sie ab; eine Schemaänderung hat wenig damit zu tun, Eigentum neu zuzuweisen, und braucht man je eine Massenneuzuweisung, ist sie ein Produktfeature, das zu bauen ist, und keine Anweisung, die man ausführt. Schreibzugriffe auf Restricted-Spalten (`iban`, `account_holder_name` und `token_value`) nehmen denselben Weg, weshalb keine Lane einem Menschen erlaubt, sie mit rohem SQL zu schreiben. Der eine Migrationsfall ist das Verschieben von Restricted-Spalten in die getrennten Secrets-Tabellen aus Kapitel 11; er geht an einen benannten Genehmiger und nicht in eine Ablehnung.

Die Spalte Migration entspricht dem, was Kapitel 11 beschreibt. Ihr Klassifikator markiert Anweisungen für einen benannten Genehmiger oder für ein Platform-Review und lehnt nur in der beschleunigten Lane ab. In dieser Tabelle erscheinen die Markierungen als Gates, bei denen eine Person Ja sagen muss, und die Allows sind die geprüfte Schemaänderung und der

Backfill, der innerhalb seiner Regel bleibt. Append-only-Tabellen verhalten sich genauso: Eine Migration, die sie berührt, geht an das Platform-Team, und die anderen Lanes verweigern sie. Ein zweites Regelwerk für verbotene Anweisungen wäre eine zweite Richtlinie, die man mit dieser abgleichen müsste, daher nutzt die Migrations-Pipeline den Klassifikator und die Matrix wieder und fügt nur ihre Lane-Regeln hinzu.

Die Matrix ist absichtlich klein. Neun Klassen und fünf Lanes passen auf eine Seite, und eine Richtlinie, die auf eine Seite passt, wird gelesen. Jede neue Klasse und jede neue Lane ist eine Entscheidung, die jemand verteidigen muss, und der übliche Fehler ist eine Matrix, die für jeden Incident eine Ausnahme bekommt, bis niemand mehr sagen kann, was sie verbietet. Ich würde sie in der Versionskontrolle halten, wie Code prüfen, mit dem Platform-Team als Verantwortlichem und dem Data-Team als Pflicht-Reviewer für die Schreibzeilen.

Die Genehmigung an die Anweisung binden

Eine Genehmigung, die „einen Schreibzugriff auf `invoices`“ abdeckt, lässt jeden Schreibzugriff auf `invoices` durch. Eine an die Anweisung gebundene Genehmigung bedeutet, dass die Genehmigung diese Anweisung benennt und das Gateway jede andere ablehnt. Kapitel 8 beschreibt den Antrag und Kapitel 10 den Runner; dieser Abschnitt ist die Regel, die beide verbindet, einmal formuliert.

Ein Antrag für einen Schreibzugriff trägt eine `fix_id` und einen `approved_hash`. Der Hash ist der Hash der Anweisungen, die geprüft wurden, über die gesamte Menge gebildet, wenn eine Korrektur mehr als eine hat, und seine Herkunft zählt mehr als die Prüfung. Er stammt aus dem Merge-Datensatz der Korrektur oder aus dem genehmigten Antragsdatensatz und nie aus der Eingabe des Runners. Lieferte der Runner seinen eigenen Hash, würde er seine eigene Arbeit prüfen, und eine Änderung an der Anweisung käme zusammen mit einem passenden Hash an. Ein Hash, der aus dem Datensatz stammt, den eine zweite Person genehmigt hat, ist ein Hash, für den jemand anderes als der Schreibende gebürgt hat.

```
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
```

Das Gateway prüft jeweils eine Anweisung, aber der Hash deckt die Menge ab: Es lädt die Anweisungen aus dem Merge-Datensatz, bestätigt, dass ihr Hash der genehmigte ist, und akzeptiert nur die nächste Anweisung in dieser Reihenfolge. Die Schranke ist die erwartete Zeilenzahl im selben Datensatz und keine Zahl, die der Antrag trägt, sodass der Hash auch die Anzahl schützt. Der Commit bleibt bei der Transaktion des Runners, sodass eine Korrektur mit mehreren Anweisungen entweder ganz oder gar nicht landet.

Zwei Regeln vervollständigen die Bindung. In einer JIT-Schreib-Session leitet das Gateway Schreibzugriffe nur von der Verbindung des Runners weiter. Der eigene Client der Person kann innerhalb der vergebenen Tripel weiterhin lesen, aber ein von ihm gesendeter Schreibzugriff wird abgelehnt, denn sonst wäre der Hash eine Prüfung, die ein Client umgehen könnte. Eine Schutzvorkehrung ist nur dann ein Mechanismus, wenn die Session auf keinem anderen Weg schreiben kann. Die zweite Regel ist der umgekehrte Fall: Ein JIT-Antrag, der nur Tripel für `SELECT` enthält, hat keine Korrektur und keinen Hash, sodass es nichts gibt, woran ein Schreibzugriff gebunden werden könnte, und der Schreibzugriff wird verweigert. Ein lesender Antrag lässt sich nicht in einen schreibenden verwandeln, indem man eine andere Anweisung sendet.

Wo das Gateway ein Credential nur vermittelt und der Client direkt verbindet, ist nichts davon durchsetzbar, und das Buch sagt das offen: Der Hash hält dann eine Absicht fest und verhindert nichts. Die Bindung ist

eine Eigenschaft des Datenpfads und existiert nur dort, wo das Gateway in ihm liegt.

Der Runbook-Fall ist der mit einer Vorlage. Der Merge-Datensatz eines Runbooks speichert den Hash der Anweisungsvorlage, und der Antrag trägt nur die Schlüsselliste. Der Runner setzt die Schlüssel in die gespeicherte Vorlage ein. Für diesen Fall prüft das Gateway den Hash gegen die Vorlage und nicht gegen den eingesetzten Text: Es bestätigt, dass die eingereichte Anweisung die genehmigte Vorlage ist, in die nur die Schlüsselliste eingetragen wurde, und verwendet die Anzahl der Schlüssel als Schranke, zusammen mit den Tripeln des Antrags. Das Gateway nimmt nie eine Anweisung oder einen Hash von dem Aufrufer, der die Arbeit erledigt haben will.

Aufrufer, die keine Menschen sind

Jobs, Runbook-Skripte, ChatOps-Bots und KI-Agenten reichen alle Anweisungen ein, und keiner von ihnen ist ein Mensch. Die Regel dieses Kapitels lautet, dass sie als eigene Principals dieselbe Schleife durchlaufen. Für Automatisierung gibt es keinen leichteren Pfad, denn Automatisierung ist der Ort, an dem sich ein Fehler mit Maschinengeschwindigkeit wiederholt.

Ein automatisierter Aufrufer hat eine Workload Identity der Plattform und nicht die Session einer Person, und einen schreibgeschützten Standard. Er öffnet seine Session unter seinem eigenen Principal, sodass seine Lesezugriffe demselben Row Cap, demselben Timeout und denselben Guarded Settings unterliegen wie die aller anderen. Will er schreiben, geht er denselben Weg wie eine Person: Er benennt eine Korrektur, ein Genehmiger, der eine andere Person ist als die, die die Automatisierung ausgelöst hat, genehmigt sie, und seine Schreibzugriffe laufen mit einer erwarteten Anzahl über den Runner. Das Vier-Augen-Prinzip aus Kapitel 8 gilt auch hier: Wer die Automatisierung gebeten hat, die Korrektur zu entwerfen, kann nicht diejenige Person sein, die sie genehmigt, und es gibt keine automatische Genehmigung durch einen Aufrufer, der kein Mensch ist. Ein Agent, der eine Anweisung vorschlägt, schlägt sie vor, und die Genehmigung ist ein Mensch, der sie liest.

Die Aufzeichnung nennt beide Beteiligten: die Automatisierung, die die Anweisung eingereicht hat, und die Person, die sie ausgelöst hat. Das ist die Hälfte der Zurechenbarkeit. Eine Aufzeichnung, die nur „der Assistent war es“ sagte, ließe die Spur bei einer Maschine enden, und eine, die nur „Sam hat darum gebeten“ sagte, würde verbergen, welches Tool die Bitte in SQL verwandelt hat. Beide Namen werden gespeichert, und das Zugriffs-Review fragt nach beiden.

Ein KI-Agent bringt ein Anliegen mit, das die anderen Aufrufer nicht haben: Seine Anweisungen werden aus Text erzeugt, von dem einiger nicht von ihm stammt. Ein Ticket, eine Log-Zeile oder eine Kundennachricht kann im Kontext landen, aus dem die Anweisung entsteht. Die Verteidigung besteht nicht darin, den Text sorgfältiger zu lesen. Sie besteht darin, dass sich die Privilegien des Agenten nicht mit dem ändern, was er liest. Seine Grants sind dieselben, seine Schreibzugriffe brauchen weiterhin die Genehmigung einer Person für die genaue Anweisung, und eine Anweisung, die auf einem Weg eintrifft, den der Agent nicht haben sollte, etwa ein Schreibzugriff außerhalb des Runners, wird von derselben Matrix abgelehnt. Wozu auch immer der Agent überredet wurde, es vorzuschlagen: Er kann es nicht genehmigen, und er kann es nicht ohne die Schranke ausführen.

Für einen nur lesenden Agenten sind die nützlichen Limits die gewöhnlichen: ein enger Row Cap, ein kurzer Timeout, eine View oder eine maskierte Kopie anstelle von Restricted-Spalten und ein Budget je Principal, sodass eine Schleife, die schiefläuft, ihr eigenes Kontingent erschöpft und nicht das gemeinsame. Für einen schreibenden Agenten ist das nützliche Limit, dass seine Schreibzugriffe wenige und klein sind, weil jeder eine Person braucht. Macht ihn das für Schreibzugriffe zu langsam, um nützlich zu sein, lautet der richtige Schluss, dass die Aufgabe zu einem Produktfeature werden sollte, das der Agent aufruft, und nicht, dass die Genehmigung gelockert werden sollte.

Abwägungen und Kosten

Vermittlung kostet Zeit. Jede Anweisung durchläuft eine Komponente mehr, und ein Schreibzugriff mit Gate wartet auf eine Person. Bei interaktiven Lesezugriffen auf einem Replikat ist die zusätzliche Latenz gering und meist unsichtbar. Bei Schreibzugriffen ist sie der Zweck: In der Verzö-

gerung findet der zweite Blick statt. Ich würde sie für dringende Fälle nicht entfernen; ich würde das Gate der Break-Glass-Lane schnell und laut machen und es im Nachhinein prüfen.

Sie kostet Kompatibilität. Ein Gateway, das Anweisungen parst und ablehnt, was es nicht parsen kann, wird manche Clients, manche Treiber und manche Gewohnheiten brechen. Das Ablehnen der Anweisung `SET`, von Zeichenketten mit mehreren Anweisungen und ungewöhnlicher Syntax erzeugt eine Liste von Dingen, die nicht mehr funktionieren. Die Liste ist kürzer, wenn sie vor dem Rollout aus den Anweisungsaufzeichnungen der bestehenden Sessions zusammengetragen wird, als wenn sie über Support-Tickets entdeckt wird. Planen Sie eine Phase ein, in der das Gateway seine Urteile aufzeichnet, aber nur die eindeutigsten Fälle ablehnt, und gehen Sie klassenweise zur Durchsetzung über, sobald die falschen Ablehnungen sinken.

Sie kostet eine Komponente, die verfügbar sein muss. Die Anweisungsprüfung sitzt im Pfad jedes Schreibzugriffs, und Kapitel 7 beschreibt, warum diese Komponente klein bleiben sollte. Ist die Prüfung ausgefallen, ist der Standard, der den Schaden begrenzt, dass Schreibzugriffe anhalten und Lesezugriffe auf dem Replikat weiterlaufen. Eine Prüfung mit Fail-open-Verhalten lehrt das Team, dass die Prüfung optional ist, und der erste Ausfall beendet die Richtlinie.

Sie kostet eine Richtlinie, die gepflegt werden muss. Die Matrix, die Liste der Guarded Settings, die Toleranz und die Row Caps sind allesamt Parameter, die jemandem gehören. Eine Richtlinie, die niemandem gehört, driftet zu dem, was der letzte Incident brauchte. Die Matrix ist das eine Artefakt, das klein, geprüft und versioniert bleiben sollte.

Es gibt außerdem Kosten, die sich nicht mit Technik bezahlen lassen: Analyse vermittelt ein falsches Gefühl von Sicherheit. Ein Team mit einem Parser, der `UPDATE` ohne `WHERE` zurückweist, kann beginnen, eine durchgelassene Anweisung als eine Anweisung zu behandeln, die keinen Schaden anrichten kann. Die Matrix sagt, was das Gateway ablehnt, und der Anfang dieses Kapitels zeigt, was eine durchgelassene Anweisung dennoch anrichten kann. Die Zählprüfung und die zweite Person gibt es für diesen Fall, und man sollte sie nicht ausdünnen, weil der Parser gut ist.

Häufige Fehler

Den Analysator als Grenze behandeln. Der Analysator ist eine Schicht, die die häufige falsche Anweisung abfängt. Die Grants entscheiden, was erreichbar ist, und ein Entwurf, der den Analysator dieses Gewicht tragen lässt, scheitert beim ersten Mal, wenn eine Anweisung gültig und falsch ist.

Den Runner oder den Aufrufer den Hash liefern lassen. Ein Hash, der mit der Anweisung eintrifft, beweist nur, dass die Anweisung zu sich selbst passt. Der Hash stammt aus dem Merge-Datensatz oder dem genehmigten Antragsdatensatz, und die Prüfung läuft im Gateway, wo der Aufrufer sie nicht bearbeiten kann.

Auto-Commit bei Schreib-Sessions eingeschaltet lassen. Bei eingeschaltetem Auto-Commit ist die Zeilenzahl ein Bericht im Nachhinein. Es ist das Ausschalten und das Anzeigen der Zahl vor dem `COMMIT`, das eine falsche Zahl billig macht.

Offen fehlschlagen, wenn der Parser eine Anweisung nicht lesen kann. Wer ausführt, was nicht analysiert werden konnte, lässt die Analyse nur die harmlosen Fälle abdecken. Lehnen Sie sie ab und führen Sie eine kurze Liste der Clients, die geändert werden müssen.

Ein leichterer Pfad für Automatisierung. Jobs und Agenten eine Abkürzung an der Schleife vorbei zu geben, weil man ihnen vertraut oder weil sie es eilig haben, entfernt die Kontrollen dort, wo das Volumen am höchsten ist. Automatisierung durchläuft dieselbe Schleife unter ihrem eigenen Principal.

Eine Matrix, die pro Incident eine Ausnahme bekommt. Jede Ausnahme ist eine Entscheidung ohne Verantwortlichen. Halten Sie die Tabelle bei den Klassen und Lanes, die es gibt, und ändern Sie den Entwurf oder bauen Sie das Feature, wenn ein Fall nicht passt.

Einen Row Cap als Limit der Gefährdung lesen. Ein Cap begrenzt eine Abfrage, nicht das, was eine Person an einem Tag lesen kann. Geht es um die Gefährdung, gehört das zur Arbeit an Maskierung und Stufen aus Kapitel 12 und Kapitel 13.

Zentrale Erkenntnisse

- Ausführung ist, wie eine Anweisung die Daten erreicht: wer sie einreicht, ob sie vorher jemand prüft, was ihre Wirkung begrenzt und wie sie mit einem Commit abgeschlossen wird. Grants beantworten diese Fragen nicht.
- Vermittelte Ausführung stellt eine Komponente zwischen die Anweisung und die Daten. Die günstigste Form sind einige durchgesetzte Einstellungen; die stärkste genehmigt jede Anweisung und führt sie von einem Worker aus.
- Jede Anweisung, von einer Person oder einer Automatisierung, durchläuft eine Schleife: Guarded Settings, Bewertung, zuerst die Aufzeichnung, ein Row Cap für Lesezugriffe und Auto-Commit aus bei Schreibzugriffen, wobei die Zeilenzahl vor dem `COMMIT` angezeigt wird.
- Analyse verringert Fehler und ist nicht die Autorisierungsgrenze. Die Grants sind die Grenze, und die in der Transaktion geprüfte echte Zeilenzahl fängt ab, was Parser und Planer nicht können.
- Eine kleine Matrix aus Anweisungsklasse und Lane gibt jede Antwort als deny, gate oder allow. Die Änderung der Eigentümerverknüpfung und Schreibzugriffe auf Restricted-Spalten laufen über ein Gate und werden auf den Produktpfad geleitet.
- Die Genehmigung ist an die Anweisung gebunden: Der `approved_hash` stammt aus dem Merge-Datensatz oder dem genehmigten Antragsdatensatz, nie vom Runner, und in einer JIT-Schreib-Session leitet das Gateway Schreibzugriffe nur vom Runner weiter.
- Jobs, Runbooks, ChatOps und KI-Agenten sind Principals mit eigener Identität, einem schreibgeschützten Standard und derselben Schleife. Ihre Aufzeichnungen nennen sowohl die Automatisierung als auch die Person, die sie ausgelöst hat.

Die Schleife und die Matrix sagen, was das Gateway mit einer Anweisung tut. Sie sagen nicht, wie man eine Korrektur schreibt, die eine Genehmigung wert ist: wie man die Anzahl belegt, einen Snapshot aufbewahrt, den Rollback plant und eine zweite Person prüfen lässt. Kapitel 10 wendet sich dieser Arbeit zu.

Kapitel 10 — Ad-hoc-Abfragen und Datenkorrekturen

Stellen Sie sich einen Donnerstagnachmittag bei Ledgerly vor. Ein Bug in `invoice-worker` ist behoben und ausgerollt, doch solange er live war, hat er eine Reihe von Rechnungen im Status `sending` zurückgelassen, obwohl die E-Mails längst verschickt waren. Der Support fragt nach, die Erinnerungen an die Kunden sind pausiert, und Robin, Backend-Entwickler in der Rufbereitschaft, sieht die Reparatur in einer einzigen Zeile vor sich: den Status der betroffenen Rechnungen auf `sent` zurücksetzen und nichts sonst.

Robin öffnet eine Session über das Access-Gateway. Die Standard-Session ist schreibgeschützt, die Zeile lässt sich also nicht ausführen. Robin muss Schreibzugriff beantragen, und der Antrag muss Tabellen und Spalten benennen. Jemand anderes muss die Anweisung ansehen, bevor sie den Primary berührt. Währenddessen läuft die Uhr, und die Reparatur ist so klein, dass der Prozess drumherum größer wirkt als die Arbeit selbst.

Dieses Kapitel fügt dem Zielzustand einen Schritt hinzu: Schreibzugriffe laufen über ein geprüftes Verfahren für Datenkorrekturen und über Just-in-Time-Grants innerhalb der Obergrenze `human_jit_write`. Kapitel 7 hat Menschen eine einzige Tür gegeben, und Kapitel 8 hat die Grants gebaut, die diese Tür für begrenzte Zeit öffnen. Hier fragen wir, was eine Person tun sollte, sobald die Tür offen ist: wie sie liest, wie sie schreibt und wie sich die zweite Art von Arbeit so absichern lässt, dass man sie an einem Donnerstagnachmittag in Ruhe erledigen kann.

Im Modell mit fünf Teilen aus Kapitel 1 geht es in diesem Kapitel um die **Ausführung**. Robin besitzt bereits jedes Credential, das die Reparatur braucht, das Risiko liegt also nicht darin, wer ein Credential hat. Es liegt darin, was der Grant erlaubt, was eine einzelne Anweisung beim Ausführen tut, wer sie vorher gelesen hat und welcher Nachweis zurückbleibt.

Die Frage dieses Kapitels lautet: Was ist der kleinste, am besten prüfbare Weg, auf dem eine Person Produktionsdaten ändern kann, und wann sollte die Änderung überhaupt nicht mehr die Aufgabe einer Person sein?

Das Problem und warum es schwierig ist

Jedes System, das echte Daten hält, gerät irgendwann in einen Zustand, den sein eigener Code nicht reparieren kann. Ein Worker ist zwischen zwei Schritten abgestürzt. Eine Migration hat eine Spalte nur halb befüllt. Ein Kunde wurde versehentlich mit einem anderen zusammengeführt. Die Anwendung hat keinen Endpoint für „das hier in Ordnung bringen“, weil niemand erwartet hat, dass sie ihn braucht, und den Endpoint zu schreiben würde länger dauern als die Reparatur. Also schreibt eine Person eine Anweisung und führt sie gegen die Produktion aus. Das ist eine **Datenkorrektur**: eine Änderung an Produktionsdaten, die eine Person außerhalb des normalen Anwendungspfads vornimmt, um einen Zustand zu reparieren, den die Anwendung nicht reparieren kann.

Die erste Schwierigkeit ist, dass das gefährliche und das nützliche Werkzeug dasselbe Werkzeug sind. Eine SQL-Session, die „Wie viele Rechnungen hängen fest?“ beantworten kann, kann mit einem weiteren Schlüsselwort alle ändern. Im Ausgangszustand aus Kapitel 1 hält der GUI-Client den gemeinsamen Admin-Login, führt jede Anweisung per Auto-Commit aus und führt aus, welcher Text gerade markiert ist. Zwischen einer Frage und einer nicht umkehrbaren Änderung liegt ein einziges Wort.

Die zweite Schwierigkeit ist, dass die Person, die die Arbeit macht, meist unter Druck steht und meist recht hat. Ein Prozess, der jede Korrektur als Bedrohung behandelt, bringt Menschen bei, ihn zu umgehen, und die Umgehung ist schlimmer als das ursprüngliche Risiko: ein Dauer-Grant, ein gemeinsames Passwort, eine Kopie der Produktion auf einem Laptop. Ein Prozess, der jede Korrektur als Routine behandelt, bringt niemandem bei, zweimal hinzusehen, und die eine falsche Anweisung unter hundert erreicht jede Zeile. Der sichere Pfad muss der einfache sein.

Die dritte Schwierigkeit ist, dass die Einheit des Risikos nicht die Person ist, sondern die Anweisung, und eine Anweisung trägt fast keine Information über ihre eigene Größe in sich. `UPDATE invoices SET status = 'sent' WHERE id = 'inv-0042'` und dieselbe Zeile ohne die Klausel

`WHERE` unterscheiden sich um ein paar Zeichen. Nichts im Text sagt „das wird neuntausend Zeilen berühren“; das sagen nur die Daten, und die Daten werden erst im Nachhinein gelesen, es sei denn, der Prozess stellt die Frage vorher.

Die vierte ist, dass die Antwort nicht allein eine Richtlinie sein kann. „Zählen Sie die Zeilen, bevor Sie aktualisieren“ wird einmal gelesen und befolgt, bis der erste hektische Tag kommt. Deshalb muss es in den Pfad eingebaut sein: ein Runner, der den Commit verweigert, wenn die Zählung nicht stimmt, und eine Session, die auf keinem anderen Weg schreiben kann.

Ad-hoc-Abfragen haben ein leiseres Problem. Eine **Ad-hoc-Abfrage** ist eine Abfrage, die eine Person schreibt und einmal ausführt, außerhalb jeder Anwendung. In einer schreibgeschützten Session kann sie nichts ändern, aber sie kann sehr viel lesen, sie kann für die Datenbank teuer in der Ausführung sein, und ihre Ergebnisse verlassen das System als Screenshot, als Tabellenkalkulation oder als Chat-Nachricht.

Der Standard: eine schreibgeschützte Session

Eine Session eines Menschen beginnt als Login, der Mitglied von `human_readonly` ist und über das Access-Gateway mit `ledgerly_replica_2` verbunden wird. Sie kann die Tabellen lesen, die die Rolle abdeckt, wobei Restricted-Spalten zurückgehalten und Confidential-Spalten dort maskiert werden, wo Kapitel 12 es vorsieht. Sie kann nicht schreiben, und nichts, was die Person eintippt, ändert das.

Die meisten Untersuchungen passen in diese Session, und die Person muss weder etwas beantragen noch auf jemanden warten. Ich würde diesen Standard allein schon aus Kostengründen verteidigen: Ein Design, in dem die häufige Aufgabe keine Genehmigung braucht, ist ein Design, in dem die Genehmigung für die seltene Aufgabe noch etwas bedeutet.

Drei Gewohnheiten halten das gemeinsame Replikat und seine Ergebnisse sicher. **Die Abfrage eingrenzen**: Der Statement-Timeout auf Rollenebene (Kapitel 5) und der Row Cap des Gateways (Kapitel 7) stoppen den Ausreißer, und wer zuerst Zählungen und Stichproben abfragt, hält versehentliche Exporte klein. **Nur das Nötigste auswählen, was die Frage beantwortet**: `id, status, due_on` ist besser als `SELECT *`, zum einen weil es

schneller ist, zum anderen weil es Confidential-Spalten aus dem Ergebnis heraushält, solange die Frage sie nicht braucht. **Ergebnisse als Datenkopien behandeln**: Ein Ergebnis, das in ein Ticket oder eine Tabellenkalkulation eingefügt wird, ist eine Kopie von Produktionsdaten unter anderen Zugriffsregeln (Kapitel 13 behandelt solche Kopien als Seitentüren). Fügen Sie also Kennungen und Zählungen ein, keine Zeilen mit personenbezogenem Inhalt, und nennen Sie die Abfrage, die sie erzeugt hat.

Eine Einschränkung ist für den nächsten Abschnitt wichtig. Das Replikat liegt um einen gewissen Betrag hinter dem Primary zurück, meist um einen kleinen und manchmal um einen größeren. Eine dort ermittelte Zählung ist eine Schätzung dessen, was der Primary jetzt enthält. Für eine Untersuchung genügt das. Um zu entscheiden, wie viele Zeilen ein Schreibvorgang berühren wird, genügt es nicht, und das Verfahren für Datenkorrekturen weiter unten ermittelt die maßgebliche Zählung auf dem Primary, innerhalb der Transaktion, die den Schreibvorgang ausführt.

Schreiben ist eine andere Handlung

Ein Lesevorgang lässt sich wiederholen, abbrechen und ignorieren. Ein Schreibvorgang ändert, was jeder andere Principal sieht, einschließlich der Services, die in diesem Moment laufen und auf den neuen Zustand reagieren, bevor jemand ihn prüft. Ein falsch aktualisierter `status` kann eine Erinnerungs-E-Mail auslösen; ein falsch aktualisiertes `due_on` kann ändern, was einem Kunden als Schuld mitgeteilt wird. Deshalb gibt das Buch dem Schreiben einen eigenen Pfad, mit eigener Genehmigung, eigener Zeitbegrenzung und eigener Aufzeichnung.

Dieser Pfad ist der Just-in-Time-Zugriff aus Kapitel 8, mit einer Eigenschaft, die hier zählt: Ein Schreibantrag ist an eine geprüfte Datenkorrektur gebunden. Der Antrag trägt die `fix_id` der Datenkorrektur und einen `approved_hash`, und seine Tripel aus (Privileg, Tabelle, Spalten) werden aus der Datenkorrektur abgeleitet, nicht vom Antragsteller getippt. Das Äußerste, worum ein Antrag bitten darf, ist die Obergrenze, die `human_jit_write` definiert. Drei Folgen bestimmen, wie die Datenkorrektur geschrieben wird.

Erstens: **Der Antrag wird aus der Datenkorrektur gebaut**. Eine Datenkorrektur, die `invoices.status` aktualisiert, braucht genau ein Tripel: `UPDATE` auf `invoices` für die Spalte `status`. Die Schutzprüfung im nächsten Abschnitt liest, bevor sie schreibt, und dieses Lesen nutzt das einfache `SELECT` auf `invoices`, das `human_readonly` bereits besitzt, also ist kein Tripel mit `SELECT` nötig. Die Ausnahme ist eine Datenkorrektur, die einen Snapshot einer Confidential-Spalte anlegt, etwa `billing_address` in `customers`: Unter `human_readonly` ist die Spalte maskiert, und ein Snapshot maskierter Werte ist für einen Rollback nutzlos. Deshalb nennt der Antrag dafür ein Tripel mit `SELECT`, und der Genehmiger sieht es.

Zweitens: **Eine Restricted-Spalte erscheint nie im Antrag**. `iban`, `account_holder_name` und `token_value` lassen sich nicht benennen, und `INSERT` und `DELETE` auf `bank_accounts` und `payment_tokens` sind nicht verfügbar, weil das Einfügen oder Löschen einer Zeile Restricted-Werte setzt oder zerstört. Eine weitere Änderung liegt innerhalb der Obergrenze und ist trotzdem keine gewöhnliche Datenkorrektur: `UPDATE` von `customer_id` auf `bank_accounts` oder `payment_tokens`. Die Spalte ist nicht Restricted, aber das Verschieben einer Zeile ändert, wessen Auszahlungskonto oder Zahlungs-Token sie ist. Die Ausführungsrichtlinie sieht diese Änderung der Eigentümerverknüpfung als Gate vor und leitet sie auf den unten beschriebenen Produktpfad um (Kapitel 9). Braucht eine Datenkorrektur eines davon, ist sie keine Datenkorrektur.

Drittens: **Der Genehmiger sieht die Anweisung, nicht nur die Tripel**. Kapitel 9 beschreibt diese Regel der an die Anweisung gebundenen Genehmigung und wie das Gateway sie durchsetzt. Bei einer Datenkorrektur zeigt der Genehmigungsbildschirm die Anweisung, den Filter und die erwartete Anzahl neben den Tripeln, und ein Antrag auf alle Spalten ist ein Antrag auf eine andere Art von Vertrauen. Datenkorrekturen mit schmalem Fußabdruck werden schneller genehmigt, und genau diesen Anreiz wollen Sie.

Der Antragsdatensatz für die Datenkorrektur der hängenden Rechnungen, in der engine-unabhängigen Notation aus Kapitel 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")
```

Bei Engines ohne Grants auf Spaltenebene liegen die Restricted-Spalten bereits in eigenen Tabellen außerhalb der Obergrenze (Kapitel 3). Ein JIT-Antrag auf die Haupttabelle ist dann ein Grant auf Tabellenebene, und der Genehmiger hat jede Nicht-Restricted-Spalte davon genehmigt, auch die Confidential-Spalten. Der Grant begrenzt den Fußabdruck der Anweisung nicht mehr, deshalb trägt die Anweisung selbst mehr von der Prüfung.

Aufbau einer Datenkorrektur

Eine Datenkorrektur ist ein kurzes Verfahren mit fester Reihenfolge. Die Reihenfolge zählt mehr als jeder einzelne Schritt, denn jeder Schritt dient dazu, den nächsten billiger richtig zu machen.

1. **Die Absicht benennen.** Ein Satz: welcher Zustand falsch ist, welcher richtig ist und welche Zeilen betroffen sind. Lässt sich das nicht in einem Satz sagen, sind es wahrscheinlich zwei Datenkorrekturen.
2. **Die betroffenen Zeilen mit einem `SELECT` zählen.** „Vierzig Rechnungen“ und „neuntausend“ sind verschiedene Gespräche, und an der Zählung zeigt sich ein Fehler zuerst.
3. **Die erwartete Zeilenzahl aufschreiben.** Die Datenkorrektur gibt die Zahl an, die sie zu ändern erwartet. Das ist die Schutzprüfung.
4. **Einen Snapshot vor der Änderung anlegen.** Die Primärschlüssel und die alten Werte genau der Spalten, die die Datenkorrektur ändern wird, an einem Ort gesichert, der die Session überdauert.
5. **Die Änderung innerhalb einer Transaktion ausführen.** Die Zahl der geänderten Zeilen mit der erwarteten Zahl vergleichen. Weichen sie

ab, Rollback.

6. **Eine Nachbedingung innerhalb derselben Transaktion prüfen.** Eine Abfrage, die beweist, dass der Zustand jetzt dem entspricht, was die Absicht sagte, zum Beispiel „keine Rechnung in der Liste hat noch den Status `sending`“.
7. **Commit, dann aufzeichnen.** Der Commit ist der Punkt, ab dem andere Principals die Änderung sehen. Der Zustand danach, innerhalb der Transaktion gelesen, kommt neben den Snapshot vor der Änderung in die Aufzeichnung.

Der Pseudocode legt die Schutzprüfung in den Runner und nicht in das Gedächtnis der Person. Er behandelt die Datenbank, den Snapshot-Speicher und das Audit-Log als getrennte Systeme, sodass keine Transaktion mehr als die Datenbank umfasst. Der Runner bekommt eine Antrags-ID und sonst nichts; die erste Funktion entscheidet, ob dieser Antrag überhaupt laufen darf.

```
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
```

Die zweite Funktion führt sie aus, und alles, was sie tut, wird von der ersten begrenzt:

```
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)
```

Mehrere Details in diesen Blöcken sind Absicht.

Der Runner liefert nie seinen eigenen Hash. Er lädt die Datenkorrektur, die der Antrag nennt, und vergleicht den Hash ihrer Anweisungen mit dem `approved_hash` im Antragsdatensatz, den Kapitel 9 auf den Merge-Eintrag zurückführt. Wer die Anweisung neu tippt, ändert den Text zwischen Prüfung und Ausführung, und die Prüfung hat sich an den früheren Text geheftet.

Der Snapshot wird vor der Transaktion gesichert. Fällt der Snapshot-Speicher aus, stoppt die Datenkorrektur dadurch, ohne dass etwas geändert wurde. Der Preis ist, dass sich die Zeilen zwischen Snapshot und

Schreibvorgang ändern könnten: Ein Worker kann in dieser Lücke eine Rechnung von `sending` auf `failed` setzen, und der Schlüssel würde weiterhin zum Filter passen.

Die Transaktion liest die Zeilen unter einer Sperre erneut. Sie vergleicht das gesamte Ergebnis, mit den Spalten, die die Datenkorrektur ändert, mit dem Snapshot. Jeder Unterschied in der Zeilenmenge oder in einem Wert bricht den Lauf mit `RowsChangedSinceSnapshot` ab, und der Autor beginnt von vorn mit einer frischen Zählung. Bei Engines, die Zeilen nicht per Filter sperren können, läuft das erneute Lesen trotzdem innerhalb der Transaktion; es verengt die Lücke und schließt sie nicht, und es bleiben die erwartete Zählung und die Nachbedingung.

Die Aufzeichnung wird nach dem Commit geschrieben. Der Snapshot-Speicher und das Audit-Log sind andere Systeme, deshalb wird der Zustand danach gesichert, sobald der Commit stattgefunden hat. Das ist die umgekehrte Reihenfolge wie beim JIT-Grant in Kapitel 8, wo ein Grant nicht ohne seinen Log-Eintrag existieren darf.

Die Reihenfolge ist hier vertretbar, weil das Gateway von Ledgerly im Datenpfad sitzt (siehe unten): Es leitet die Anweisungen des Runners weiter und zeichnet die Session auf, sodass eine festgeschriebene Änderung auch dann dokumentiert ist, wenn das Anhängen an das Audit-Log fehlschlägt. Der Eintrag wird erneut versucht, und das Platform-Team wird benachrichtigt. Bei einem Gateway, das nur Credentials vermittelt, gäbe es kein Session-Log der Anweisungen, und der Audit-Eintrag müsste zuerst kommen.

Ein Dry Run zeigt, was eine Zählung nicht zeigen kann. Die Zählung ist ein `SELECT`, sie übersieht also, was die Anweisung in Gang setzt: einen Trigger, der anderswo schreibt, eine Kaskade, eine Regel. Ein echter Dry Run führt die Änderung innerhalb einer Transaktion aus und macht sie per Rollback rückgängig:

```
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
```

Der Bericht enthält die Zahl der Zeilen, eine Stichprobe der Werte danach und die Veränderung der Zeilenzahlen in den `watch_tables` aus dem Header der Datenkorrektur. Er braucht einen genehmigten Antrag und die Schreib-Grants, kann also nicht existieren, solange der Pull Request in der Prüfung ist; er läuft innerhalb des genehmigten Zeitfensters, hält dabei dieselben Zeilensperren und kann Effekte außerhalb der Datenbank, etwa eine E-Mail, nicht sehen. Wo eine Überraschung teuer wäre, führt der Autor ihn in den ersten Minuten des Zeitfensters aus und hängt den Bericht an den Antrag, und der Genehmiger liest ihn vor dem Commit-Schritt, um den Preis einer Pause.

Eine Schutzprüfung ist nur dann ein Mechanismus, wenn die Session auf keinem anderen Weg schreiben kann. Das Gateway von Ledgerly sitzt im Datenpfad (Kapitel 7) und leitet in einer JIT-Schreib-Session Schreibvorgänge nur von der Verbindung des Runners weiter, sodass sich die Hash-Prüfung, die Zählung und der Snapshot nicht überspringen lassen. Die Transaktion bleibt die des Runners: Das Gateway leitet jede Anweisung in sie hinein und zählt die Zeilen über die gesamte Datenkorrektur, schreibt aber nie im Namen des Runners fest, weshalb eine fehlgeschlagene Nachbedingung jede Anweisung zurückrollt. Kapitel 9 behandelt, wie das Gateway jede Anweisung gegen den Merge-Eintrag prüft und wo seine Inspektion endet. Der Preis ist, dass jeder JIT-Schreibvorgang davon abhängt, dass der Runner und das Gateway verfügbar sind. Bei einem Gateway, das nur das Credential vermittelt, ist der Runner eine Konvention und kein Schutz gegen eine Anweisung ohne die Klausel `WHERE`.

Wohin der Snapshot gehört

Der Snapshot vor der Änderung ist eine Datenkopie und braucht, wie jede andere, eine verantwortliche Stelle und ein Ablaufdatum. Ein Snapshot von Werten der Spalte `status` in `invoices` enthält Internal-Daten. Ein Snapshot einer Confidential-Spalte, etwa `billing_address` in `customers`, ist Confidential, und der Speicher braucht Zugriffsregeln, die mindestens so streng sind wie die der Quelltabelle.

Ich würde Snapshots zusammen mit der Aufzeichnung der Datenkorrektur aufbewahren, mit einer Aufbewahrungsfrist, die das Platform-Team festlegt: lang genug, damit sich „War diese Datenkorrektur richtig?“ beantworten lässt, kurz genug, damit der Snapshot nicht zu einer zweiten, unkontrollierten Kopie von Kundendaten wird. Das sind Tage bei Rechnungsstatus und vielleicht ein volles Abrechnungsintervall bei einer Rechnungsadresse. Wählen Sie eine Zahl, schreiben Sie sie auf und lassen Sie sie ablaufen.

Der Rollback-Plan

Ein Snapshot vor der Änderung ist kein Rollback-Plan, solange niemand aufgeschrieben hat, wie er verwendet würde. Der Plan besteht aus ein oder zwei Sätzen im Header der Datenkorrektur, etwa „`status` für dieselben Schlüssel aus dem Snapshot wiederherstellen“, und es sollte möglich sein, den Snapshot mechanisch in ein Korrekturskript zu verwandeln.

Der Rollback ist selbst eine Datenkorrektur. Er durchläuft denselben Antrag und dieselbe Prüfung, denn andere Principals könnten in der Zwischenzeit auf den neuen Zustand reagiert haben, und das Wiederherstellen der alten Werte könnte berechtigte Aktualisierungen überschreiben. Der Runner für einen Rollback prüft deshalb, ob die aktuellen Werte noch dem Zustand danach entsprechen, den die ursprüngliche Datenkorrektur aufgezeichnet hat, und weigert sich, Zeilen anzufassen, die sich inzwischen geändert haben. Eine Rückwärts-Datenkorrektur, die nicht sagen kann „diese Zeilen haben sich weiterentwickelt, behandeln Sie sie getrennt“, verursacht die nächste Reparatur.

Manche Änderungen haben keinen Rollback, und der Header sollte das sagen. Eine E-Mail, ein Webhook oder ein Aufruf an einen Zahlungsanbieter lässt sich nicht zurücknehmen, indem man eine Spalte wiederherstellt. Bei

ihnen findet das Gespräch statt, bevor die Datenkorrektur läuft: Welche Effekte löst diese Zustandsänderung aus, und sollten die Services, die darauf reagieren, zuerst angehalten werden?

Zwei Tabellen haben keinen Rollback im üblichen Sinn, weil keine menschliche Rolle sie bearbeiten kann. `ledger_entries` und `audit_events` sind Append-only. Eine Korrektur am Hauptbuch ist ein neuer Eintrag, der ausschließlich über den Korrekturpfad von `payments-service` ausgestellt wird, nie über einen Login eines Menschen. Eine Datenkorrektur, die dort eine Änderung zu brauchen scheint, ist also ein Antrag auf diesen Pfad und nicht auf einen Grant. Ein Trigger, der jedes `UPDATE` und `DELETE` auf beiden Tabellen ablehnt, sorgt dafür, dass die Regel auch dann gilt, wenn ein Grant versehentlich erweitert wird. Das hat einen Preis: `ledgerly_owner` kann den Trigger weiterhin löschen, der eigentliche Prüfpunkt bleibt also bei den Migrationen, die diese Tabellen berühren (Kapitel 11).

Ein DELETE ist dasselbe Verfahren mit schwierigerem Rollback

Der obige Aufbau ist für `UPDATE` geschrieben. Angenommen, ein Retry in `invoice-worker` hat doppelte Zeilen in `invoice_lines` eingefügt, und die Reparatur entfernt sie. `DELETE` auf `invoice_lines` liegt innerhalb der Obergrenze von `human_jit_write` und unterscheidet sich in drei Punkten.

Der Grant begrenzt die Tabelle und nicht die Zeilen, denn ein Tripel mit `DELETE` hat keine Spalten. Die Begrenzung kommt also aus dem Filter, der erwarteten Zählung und der Prüfung, und der Filter sollte eine Liste von Schlüsseln sein. Der Snapshot ist die ganze Zeile, da danach nichts anderes die Werte mehr hält, und bei einer Tabelle mit Confidential-Spalten ist der Snapshot so sensibel wie die Tabelle. Der Rollback ist ein erneutes Einfügen, also eine weitere Datenkorrektur mit eigenem Antrag. Ein Tripel mit `INSERT` nennt seine Spalten, daher entscheidet der Antrag für das erneute Einfügen, ob `id` und `created_at` dazugehören. Gehören sie nicht dazu, erhält die wiederhergestellte Zeile für beide neue Werte, und alles, was auf die alten Kennungen verwiesen hat, wird nicht wiederhergestellt. Gehören sie dazu, sieht der Genehmiger, dass ein Schlüssel und ein Zeitstempel geschrieben werden, was keine gewöhnliche Datenkorrektur tut,

und was die Engine mit einem mitgelieferten Schlüssel macht, hängt davon ab, wie sie Schlüssel erzeugt. Der Header gibt an, welche Wahl der Rollback trifft.

```
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")
```

Wo der Verlust der Kennungen nicht hinnehmbar ist, verlangt der Reviewer einen billigeren Schutz als den Rollback: einen Fremdschlüssel, der das Löschen ablehnt, solange eine andere Zeile noch auf die Position verweist, oder eine andere Reparatur, die die Zeile nicht entfernt.

Review: Vier-Augen-Prinzip in einem Repository

Eine Datenkorrektur ist Code, der einmal läuft, deshalb bekommt sie die Kontrolle, der Entwickler bereits vertrauen: einen Pull Request. Die Datenkorrektur ist ein Skript in einem Repository, mit Header, Reviewer und Historie, und die Historie beantwortet sechs Monate später, wer einen Status geändert hat und warum.

```
// 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"
```

Das Feld `privileges` verbindet das Repository mit Kapitel 8. Die Tripel des Antrags werden daraus kopiert, sodass der Genehmiger Antrag und Datenkorrektur nebeneinander ansieht, und `loadApprovedFix` lehnt einen Antrag ab, dessen Tripel von der Datenkorrektur abweichen. Eine Anweisung, die eine Spalte berührt, die ihre Privilegien nicht nennen, scheitert zur Laufzeit, und das ist das beabsichtigte Ergebnis; der Grant ist die zweite Verteidigungslinie hinter der Prüfung.

Was der Reviewer liest

Die Aufgabe des Reviewers ist enger gefasst als „Ist das eine gute Idee?“.

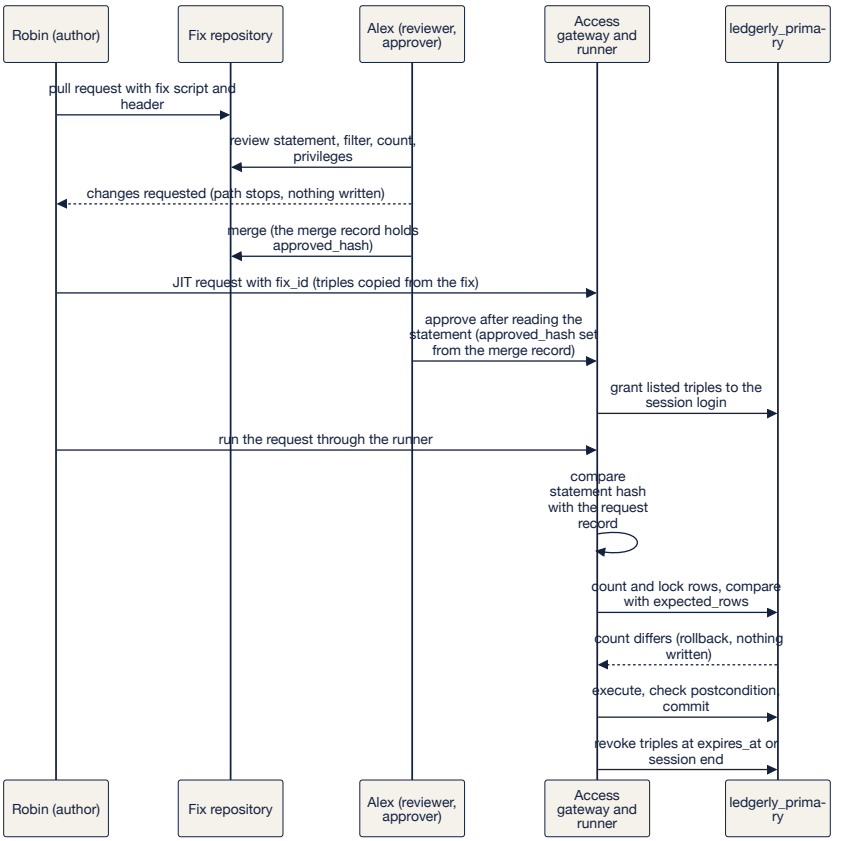
- **Passt die Anweisung zur Absicht?** Sie sollten in den Tabellen, den Spalten und dem Filter übereinstimmen.
- **Ist der Filter spezifisch?** Ein Filter, der nach Schlüsseln auswählt, ist am sichersten; ein Filter, der nach Zustand und einem Zeit-Cutoff auswählt, ist akzeptabel, wenn die Zählung stimmt; ein Filter ganz ohne Einschränkung wird auf den ersten Blick abgelehnt.
- **Ist die erwartete Zählung plausibel, und woher stammt sie?** Ein Reviewer, der `expected_rows: 40` sieht, kann die Ausgabe der Zähl-Abfrage im Pull Request verlangen. Eine Zahl, die niemand belegen kann, ist eine Zahl, die niemand geprüft hat.
- **Nennen die Privilegien nur, was die Anweisung braucht?** Vergleichen Sie sie Zeile für Zeile.

- **Berührt die Datenkorrektur eine Restricted-Spalte, eine Append-only-Tabelle oder eine Tabelle außerhalb der Obergrenze?** Wenn ja, ist sie keine Datenkorrektur, und die Prüfung endet mit einem Hinweis auf den richtigen Pfad.
- **Was löst diese Änderung aus?** Erinnerungen, E-Mails, Jobs, die die Spalte lesen. Diese Frage fängt die leisen Fälle ab.
- **Stimmt die Vorschau mit der Absicht überein?** Der Pull Request enthält die Ausgabe der Zähl-Abfrage und eine Vorschau der Zeilen und der Werte danach per `SELECT`; der Dry-Run-Bericht wird dem Antrag später angehängt, und ein Trigger oder eine Kaskade zeigt sich dort und nirgendwo sonst.
- **Ist der Rollback-Satz brauchbar?** Bei einem `DELETE` sagt er, was das erneute Einfügen nicht wiederherstellen kann.

Die Prüfung ist Vier-Augen-Prinzip im Sinne dieses Buches: Eine zweite Person prüft, bevor eine riskante Handlung läuft. Der Merge legt den Text fest: Eine Änderung am Skript nach der Prüfung erzeugt einen anderen Hash, und der Runner lehnt sie ab. Der Genehmiger des JIT-Antrags und der Reviewer der Datenkorrektur sind zwei Handlungen. In einem großen Team würde ich zwei Personen bevorzugen, damit der Genehmiger den Zugriff gegen die Datenkorrektur prüft, statt sie noch einmal zu lesen; in einem kleinen Team sind sie oft dieselbe Person, was akzeptabel ist, solange keiner von beiden der Autor ist. Der Autor genehmigt nie seine eigene Datenkorrektur oder seinen eigenen Grant.

Der Ablauf

Das Diagramm zeigt den Pfad von einem Antrag bis zu einer festgeschriebenen Änderung. Achten Sie auf die Stellen, an denen der Pfad enden kann, ohne dass etwas geschrieben wurde: die Prüfung, den Hash-Vergleich und die Schutzprüfungen des Runners.



Wenn die Änderung eine Restricted-Spalte betrifft

Die Obergrenze verbietet Restricted-Spalten in jedem JIT-Antrag (Kapitel 3 nennt den Grund), sodass für eine ganze Klasse von Reparaturen kein SQL-Pfad existiert. Angenommen, der `account_holder_name` eines Kunden in `bank_accounts` wurde mit einem Tippfehler erfasst. Eine Person kann das in einer JIT-Session nicht beheben, weil sich die Spalte in einem Antrag nicht benennen lässt. Auch eine Break-Glass-Session kann es nicht: `human_breakglass` hat kein Schreibprivileg auf Restricted-Spalten, was auch immer der Prozess sagt.

Der Pfad ist die Anwendung. `admin-console` bittet `payments-service`, den einzigen Writer von `bank_accounts`, die Änderung vorzunehmen; `admin-console` verlangt, dass eine zweite Person sie genehmigt, bevor `payments-service` sie ausführt, und die Änderung wird in `audit_events` aufgezeichnet. Das ist ein Produktpfad und keine Datenkorrektur: Er hat

ein Formular, eine Validierung, ein Begründungsfeld, einen Genehmiger und eine Aufzeichnung, und er läuft mit der eigenen Rolle des Services, nicht mit der eines Menschen.

Das kostet mehr als eine SQL-Anweisung, und ich finde, das soll es auch. Diese Werte sind diejenigen, deren Offenlegung Kunden auf schwer umkehrbare Weise schadet, und die Anwendung ist der einzige Ort, an dem Validierung und die gesamte Menge der Seiteneffekte leben. Der Preis sind Minuten für die Person, die die Änderung braucht; der Gewinn ist, dass eine von Hand getippte Reparatur keine fehlerhafte Bankkontonummer mehr einführen kann. Ein falscher Token in `payment_tokens` wird ebenso von `payments-service` über den eigenen Ablauf des Anbieters ersetzt und nicht nachgebessert.

Break-Glass ist kein Weg für Datenkorrekturen

`human_breakglass` schreibt dort, wo die JIT-Obergrenze nicht hinreicht, und ist die einzige menschliche Rolle, die Restricted-Spalten liest (ihre genauen Privilegien stehen in Kapitel 3). Auf dem Papier sieht sie wie ein bequemer Weg aus, eine Reparatur durchzuführen, die der JIT-Pfad ablehnt. Das ist sie nicht, und der Grund ist, was der Rolle fehlt.

Break-Glass ist selbst erklärt. Es hat keinen Genehmiger, keine `fix_id` und keinen Runner, und seine Schutzvorkehrungen sind das Logging-Set und die verpflichtende Prüfung, die Kapitel 8 definiert. Eine eingrenzende Handlung, die anhaltenden Schaden stoppt und nachträglich geprüft wird, ist wofür die Rolle da ist (Kapitel 15 behandelt die Seite der Incidents). Die dauerhafte Reparatur der Daten ist eine andere Handlung und immer eine Datenkorrektur: In einer Break-Glass-Session ausgeführt, überspringt sie die eine Kontrolle, die die Anweisung liest, bevor sie läuft, in einer Rolle, in der ein `UPDATE` ohne Zeilen-Schutzprüfung möglich ist. Kapitel 8 nennt das ein akzeptiertes Risiko, und die Reparaturen über die Rolle zu leiten würde es zur Routine machen.

Der zweite Grund sind die Anreize. Wenn Break-Glass eine Reparatur ausführen kann, wird es zur Überholspur, und sein Alarm verliert jede Bedeutung. Eine Datenkorrektur, die Break-Glass „braucht“, deutet auf etwas anderes hin: Der nächtliche Genehmigungszyklus ist zu langsam, der Obergrenze fehlt eine Spalte, die die Datenkorrektur berechtigterweise braucht, oder die Datenkorrektur gehört auf den Anwendungspfad. Kann

eine Reparatur nicht warten, stoppen Sie den Schaden mit der kleinsten verfügbaren Handlung und führen die Datenkorrektur auf dem normalen Pfad aus, sobald der Druck weg ist.

Eine wiederholte Datenkorrektur in ein Produktfeature verwandeln

Eine Datenkorrektur ist ein Symptom: Ein Zustand ist erreichbar, den die Anwendung nicht rückgängig machen kann. Das erste Auftreten ist eine Datenkorrektur; das dritte ist ein Feature-Wunsch, den niemand gestellt hat. Ich würde ein Verzeichnis der Korrekturskripte führen, mit Tags nach Art der Reparatur versehen, und wenn dasselbe Tag wieder auftaucht, fragt jemand, ob es nach `admin-console` gehört. An „dreimal“ ist nichts Genaues: Eine Reparatur von zehn Zeilen, zweimal im Jahr, rechtfertigt vielleicht nie einen Bildschirm, während eine, die jede Woche läuft, jedes Mal eine Prüfung braucht und vom Support in einer Minute erledigt werden könnte, ein Feature ist, das darauf wartet, geschrieben zu werden.

Nehmen Sie die hängenden Rechnungen vom Anfang und angenommen, der Zustand tritt erneut auf. Die Grant-Tabelle aus Kapitel 3 zeigt bereits, was `app_admin_console` mit `invoices` tun darf: `SELECT` und `UPDATE` nur auf `status`. Die Produktversion ist eine Aktion auf der Rechnungsseite, verfügbar für eine enge Gruppe, die den Status für eine einzelne Rechnung zurücksetzt, eine Begründung verlangt und die Aktion mit der Identität des menschlichen Principals in `audit_events` schreibt. Sie bringt ihre eigenen Prüfungen mit: nur Rechnungen im hängenden Zustand, nur eine auf einmal und eine Obergrenze dafür, wie viele eine Person pro Stunde zurücksetzen kann.

Die Umwandlung verlagert das Risiko von der Anweisung, die bei jeder Reparatur einen Antrag, eine Prüfung und einen Runner bezahlt, hin zum Code, der einmal geprüft, getestet und versioniert wird. Der Support kann die Aktion ohne Datenbank-Session nutzen, sodass jede Reparatur, die zum Feature wird, ein wiederkehrender Schreibvorgang ist, der keinen menschlichen Zugriff mehr braucht.

```
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()
```

Das Audit-Ereignis und die Statusänderung liegen beide in `ledgerly_primary`, eine einzige Transaktion deckt sie also ab; `audit_events` ist die fachliche Aufzeichnung der Anwendung, nicht das Audit-Log aus Kapitel 14. Im Runner oben ist das Audit-Log ein getrenntes System, und die Reihenfolge der Operationen trägt die Sicherheit.

Nicht jede Reparatur sollte ein Feature werden. Ein Button, den jeder drücken kann, hat ein anderes Risikoprofil als ein Skript, das zwei Personen geprüft haben, deshalb bleiben Reparaturen, die Geld, Tokens oder Bankdaten ändern, auf einem Vier-Augen-Pfad, auch wenn sie häufig sind. Eine Reparatur wird nur dann zur Self-Service-Aktion, wenn ihr schlimmster Fall durch das Design der Aktion begrenzt ist; andernfalls wird sie zu einem Antrag mit einer Genehmigung innerhalb des Produkts. Der Test ist, ob die Reparatur wiederkehren wird und ob derzeit nur eine Person sie durchführen kann. Lautet beides ja, ist der Sprint eine Investition und keine Ausgabe.

Runbooks für die Reparaturen, die Sie vorhersehen können

Manche Datenkorrekturen kehren der Art nach wieder, auch wenn sich die Einzelheiten ändern: eine Rechnung dem richtigen Kunden neu zuordnen, einen fehlgeschlagenen Export wiederholen, eine doppelte Zeile reparieren. Dafür ist ein **Runbook** das richtige Artefakt: ein aufgeschriebenes, befolgtes Verfahren für eine wiederholbare Aufgabe. Es hat dasselbe Grundgerüst wie ein Korrekturskript, mit leer gelassenen Parametern. Es nennt das Symptom, das es auslöst, die Diagnose-Abfragen, die es bestätigen, die Vorlage der Datenkorrektur und die Privilegien, die sie braucht, die Quelle der erwarteten Zählung, die zu prüfenden Effekte, den Rollback und wer schreiben und prüfen darf. Es sagt auch, wann man anhalten und eskalieren soll: „Wenn die Diagnose-Zählung die Zahl in diesem Absatz übersteigt, ist das nicht das bekannte Problem.“

Ein Runbook kann auch eine Vorab-Genehmigung tragen, damit eine vorhersehbare Reparatur nachts nicht auf einen Genehmiger wartet. Fünf Bedingungen entscheiden darüber, und der Runner prüft sie, nicht die Person:

- der Filter wählt nach Schlüsseln aus, einer Liste von IDs, und nicht nach einem Zustand und einem Cutoff;
- die erwartete Zeilenzahl beträgt höchstens N, eine Zahl, die das Runbook für seine eigene Reparatur festlegt (etwa 20 bei hängenden Rechnungen; illustrativ);
- die Tripel sind genau der feste Satz des Runbooks;
- die Reparatur löst keinen ausgehenden Effekt aus, wie das Feld `effects` des Runbooks festhält;
- die Anweisung ist die Vorlage des Runbooks, in die nur die Schlüsselliste eingesetzt wurde, und der Hash der Vorlage wurde genehmigt, als das Runbook geprüft wurde.

Der Merge-Eintrag des Runbooks speichert diesen Vorlagen-Hash. Die Schlüsselliste ist der einzige Teil, den der Antrag trägt; der Runner setzt sie in die gespeicherte Vorlage ein und prüft das Ergebnis gegen die Tripel des Antrags, und er nimmt von seinem Aufrufer nie einen Hash oder eine

Anweisung entgegen (Kapitel 9 behandelt, wie das Gateway die eingesetzte Anweisung prüft). Schlägt eine Bedingung fehl, fällt der Antrag auf die volle Prüfung mit einem Genehmiger zurück.

Der Preis ist, dass ein vorab genehmigter Lauf ausgeführt wird, ohne dass jemand in dieser Nacht die Schlüsselliste liest. Die Zeilengrenze des Runbooks begrenzt also eine falsche Liste, und am nächsten Werktag liest jemand anderes als der Runner die Aufzeichnung. Ein Runbook altert außerdem: Eines für eine Tabelle, die inzwischen ihre Form geändert hat, gibt einer veralteten Anweisung den Anschein einer Prüfung. Binden Sie jedes Runbook an ein Datum und eine verantwortliche Person, und behandeln Sie einen fehlgeschlagenen Lauf als Anlass, es zu überarbeiten, nicht als Ärgernis, das man umgeht.

Ein UPDATE ohne WHERE, ohne Schuldzuweisung analysiert

Betrachten Sie Ledgerly in seinem Ausgangszustand. Ein Entwickler verbindet sich mit einem Desktop-Client, der das gemeinsame Admin-Passwort speichert und automatisch mit einem Commit abschließt. Die Aufgabe sind die hängenden Rechnungen. Die Anweisung steht auf zwei Zeilen im Editor: das `UPDATE` auf der ersten, das `WHERE` auf der zweiten. Der Client führt die markierte Auswahl aus, und die Auswahl umfasst nur die erste Zeile. Die Anweisung läuft, und jede Zeile in `invoices`, sagen wir neuntausend, hat den Status `sent`.

Eine Version dieser Geschichte endet damit, dass jemandem gesagt wird, er solle vorsichtiger sein. Das lehrt nichts, denn der nächste Entwickler auf demselben Stuhl steht vor demselben Abstand zwischen einer Frage und einer nicht umkehrbaren Änderung. Die nützliche Frage lautet, was am Design es einer einzigen Anweisung erlaubte, jede Zeile zu erreichen.

- **Der Login konnte überall schreiben.** Die gemeinsame Admin-Rolle besitzt jede Tabelle, der Fehler war also nicht auf die Tabelle oder die Spalte begrenzt, die die Aufgabe brauchte.
- **Der Client hat von selbst einen Commit ausgeführt.** Niemand sah „neuntausend Zeilen betroffen“, bevor die Anweisung endgültig wurde.

- **Nach der Zählung wurde nie gefragt.** Nichts verlangte von der Person zu wissen, wie viele Zeilen zu erwarten waren, sodass die Zahl, die hinterher ausgegeben wurde, nichts bedeutete.
- **Niemand sonst hatte die Anweisung gelesen.** Der einzige Leser war der Autor, der die Absicht in den Text hineinlas.
- **Es gab keinen Snapshot vor der Änderung.** Die alten Status waren weg, und die Reparatur der Reparatur hatte keinen Ausgangspunkt.
- **Das Log sagte „die gemeinsame Rolle".** Die Frage „Wer hat das ausgeführt?" hatte nur über den Chatverlauf eine Antwort.

Die Wiederherstellung beginnt bei einem Backup, gleicht ab, was seither passiert ist, und informiert alle nachgelagerten Stellen (Kapitel 13 und Kapitel 15 behandeln diese Mechanik). Die Kosten hatte das Design festgelegt, bevor die Anweisung getippt wurde.

Spielen Sie nun denselben Nachmittag im Zielzustand durch. Robin schreibt die Datenkorrektur mit ihrem Header und einer erwarteten Zählung von 40, und Alex prüft sie. Der JIT-Antrag nennt `UPDATE` auf `invoices`, Spalte `status`, und der Grant gilt eine Stunde. Wäre der Filter aus dem geprüften Skript herausgefallen, würde der Runner neuntausend gegen erwartete 40 zählen und ablehnen, bevor eine Anweisung lief. Eine Anweisung, die in den eigenen Client der Person getippt wird, würde nicht als Schreibvorgang weitergeleitet (die Gateway-Regel aus Kapitel 9), und in einer einfachen Session mit `human_readonly` würde das `UPDATE` gar nicht laufen.

Die Schutzprüfung macht eine Datenkorrektur nicht richtig. Sie macht eine Klasse von Fehlern laut: Die Anweisung hat eine andere Anzahl von Zeilen berührt, als der Autor gesagt hat. Eine Anweisung, die genau 40 Zeilen berührt, aber die falschen 40, passiert sie, weshalb die Prüfung, die Nachbedingung und der Snapshot als getrennte Schichten existieren und weshalb Kapitel 9 sagt, was die Analyse nicht sehen kann. Die Schutzprüfung ist außerdem nur so gut wie ihre Zahl. Eine erwartete Zählung, die aus der Ausgabe des Laufs selbst kopiert wurde, bringt dem Runner bei, sich selbst zuzustimmen; die Zahl muss also aus einer Abfrage stammen, die der Reviewer unabhängig ausführen kann, idealerweise bevor die Anweisung geschrieben wurde. Wo die Engine Grants auf Spaltenebene hat (Kapitel 3), begrenzt ein schmaler Grant, was die Fälle erreichen können, die die Schutzprüfung verfehlt.

Abwägungen und Kosten

Das Verfahren ist langsamer als das Tippen einer Anweisung. Ein Antrag, eine Prüfung, eine Genehmigung und ein Runner kosten an einem guten Tag Minuten und sehr viel mehr, wenn der Genehmiger schläft. An einem Donnerstagnachmittag ist das akzeptabel. Nachts, unter einem Alarm, ist es der Preis, der Menschen zu Break-Glass verleitet, und er verdient Aufmerksamkeit im Design: einen Reviewer in der Rotation, die oben beschriebene Vorab-Genehmigung per Runbook für die Reparaturen, die sich dafür eignen, und eine festgelegte Obergrenze dafür, wie lange das Team auf eine nächtliche Genehmigung höchstens wartet, aufgeschrieben, bevor die erste Nacht kommt, in der man sie braucht.

Die Prüfung kann zum Reflex werden. Eine Genehmigung, die in Sekunden erteilt wird, fügt Reibung hinzu, ohne die Gefährdung zu senken; Kapitel 8 führt das Argument vollständig aus. Das Signal, das für dieses Kapitel spezifisch ist, ist der Pull Request: Wenn Reviewer nie die Ausgabe der Zähl-Abfrage verlangen und nie eine Änderung an einer Anweisung anfordern, lesen sie nichts oder prüfen nur perfekte Datenkorrekturen. Die Verteidigung sind die Checklistenpunkte, die ein Artefakt brauchen: die Zähl-Ausgabe und die Vorschau im Pull Request, der Dry-Run-Bericht am Antrag und ein ausdrücklicher Vergleich der Anweisung mit den Privilegien.

Transaktionen halten Sperren. Der Runner startet die Transaktion nach der Prüfung und beendet sie nach Sekunden. Ein Verfahren, das sie zuerst öffnet, „sicherheitshalber“, und den Menschen danach entscheiden lässt, lässt andere Writer warten, füllt die Verbindungen des Poolers und lässt die Timeouts aus Kapitel 5 bei unbeteiligter Arbeit auslösen. Halten Sie die menschliche Entscheidung außerhalb der Transaktion.

Große Datenkorrekturen brauchen Batches. Eine Datenkorrektur, die Millionen von Zeilen berührt, kann nicht als eine Transaktion laufen, ohne lange gehaltene Sperren und ein großes Transaktionslog. Die übliche Antwort ist, sie in Batches aufzuteilen, sagen wir von einigen tausend Zeilen (eine illustrative Größe, keine Empfehlung), jeder mit eigener Zählung und eigener Schutzprüfung, mit einem Batch-Modus des Runners, der dieselben Prüfungen wiederholt. Ein Fehler mittendrin hinterlässt die Daten in einem gemischten Zustand, deshalb muss die Datenkorrektur so geschrieben sein, dass sie sich erneut ausführen lässt: Der Filter wählt nur

Zeilen aus, die noch im alten Zustand sind. Eine große Reparatur ist trotzdem eine Datenkorrektur; nur die erste Befüllung einer neuen Struktur gehört zu einer Migration (Kapitel 11).

Snapshots und das Repository der Korrekturen brauchen eine verantwortliche Stelle. Ein Snapshot ohne Ablaufdatum hat ein Risiko gegen ein anderes getauscht, und bei Confidential-Spalten kann das zweite das größere sein. Ein Verzeichnis von Korrekturskripten ist eine Aufzeichnung davon, wie die Produktion abgedriftet ist, und ohne einen Index nach Tags sucht niemand nach dem früheren Auftreten eines Problems.

Wann man das alles nicht einsetzt. In einem kleinen Team mit einer Datenbank, die nichts Sensibles enthält, ist ein Pull Request für jede Statusänderung mehr Zeremonie, als das Risiko rechtfertigt; erst zählen und dann eine Transaktion verwenden, lohnt sich trotzdem. Auf einer Entwicklungs- oder Staging-Datenbank ohne Produktionsdaten gilt nichts davon, und bei einer Kopie, die jede Nacht neu aufgebaut wird, etwa `ledgerly_analytics`, kostet ein fehlerhafter Schreibvorgang einen Neuaufbau. Das Verfahren skaliert mit den Kosten eines Fehlers, und der Primary ist der Ort, an dem diese Kosten am höchsten sind.

Häufige Fehler

- **Die Klausel `WHERE` als Schutzprüfung behandeln.** Das ist verlockend, weil in der Klausel die Absicht des Autors steckt. Aber eine Klausel ist Text, und Text kann abgeschnitten, verschrieben oder teilweise ausgeführt werden. Die Abhilfe ist eine Schutzprüfung, die die Zahl der geänderten Zeilen mit einer angegebenen Erwartung vergleicht und bei Abweichung zurückrollt.
- **Auf dem Replikat zählen und auf dem Primary aktualisieren.** Das Replikat hinkt hinterher, die beiden Zahlen können also aus gutem Grund abweichen. Die Abhilfe ist, die maßgebliche Zählung auf dem Primary innerhalb der Transaktion zu ermitteln und die Zählung des Replikats nur zum Erkunden zu verwenden.
- **Die geprüfte Anweisung neu tippen.** Der Reviewer hat eine Datei genehmigt, und jemand hat sie in einen Editor kopiert und „wo man schon dabei ist“ eine Kleinigkeit geändert. Die Abhilfe ist ein Runner, der den Hash des Skripts mit dem `approved_hash` im Antragsdaten-

satz vergleicht, zusammen mit einem Gateway, das die Schreibvorgänge der Session nur von diesem Runner weiterleitet. Bei einem Gateway, das nur Credentials vermittelt, ist die Schutzprüfung eine Konvention, und die Prüfung muss sie tragen.

- **Einen weiten Grant „sicherheitshalber“ beantragen.** Er macht aus der Genehmigung ein Urteil über Vertrauen statt einer Prüfung gegen eine Anweisung, und er weitet aus, was ein Fehler erreichen kann. Die Abhilfe ist, die Tripel zu beantragen, die die Anweisung braucht, und einen zweiten Antrag zu stellen, wenn sich die Lage ändert.
- **Break-Glass nutzen, weil der Genehmiger langsam ist.** Der Pfad ist kürzer, und die Datenkorrektur ist dringend. Die Abhilfe ist, die langsame nächtliche Genehmigung zu lösen und Break-Glass für die Notfälle zu reservieren, für die es definiert wurde.
- **Eine Restricted-Spalte über eine „besondere“ Session schreiben.** Eine `iban` direkt zu aktualisieren fühlt sich wie ein Gefallen für den Kunden an, und es entfernt Validierung, Genehmigung und die Aufzeichnung der Anwendung in einem Schritt. Die Abhilfe ist der Anwendungspfad, auch wenn er langsamer ist.
- **Einen Snapshot sichern, den niemand steuert.** Der Snapshot schützt vor einer schlechten Datenkorrektur und wird zu einer unkontrollierten Kopie von Confidential-Daten in einem gemeinsamen Ordner. Die Abhilfe ist eine verantwortliche Stelle, eine Zugriffsregel, die der Quelltabelle entspricht, und ein Ablaufdatum.
- **Den Rollback-Satz auslassen, weil die Datenkorrektur klein ist.** Kleine Datenkorrekturen sind diejenigen, die in Eile rückgängig gemacht werden. Die Abhilfe ist ein einzeiliger Plan, vor dem Lauf geschrieben, der sagt, was der Snapshot wiederherstellt und wo er es nicht darf.
- **Eine wiederholte Datenkorrektur für immer als Skript lassen.** Das Ergebnis ist ein wiederkehrender Schreibvorgang, der jedes Mal einen Menschen auf dem Primary braucht. Die Abhilfe ist ein regelmäßiger Blick auf die Tags im Korrekturverzeichnis und eine Entscheidung, auch wenn die Entscheidung „noch nicht“ lautet.
- **Auf die Richtigkeit der Absicht statt auf die Richtigkeit der Anweisung prüfen.** Ein Reviewer, der darüber streitet, ob die Reparatur

überhaupt sinnvoll ist, liest die Anweisung womöglich nie. Die Abhilfe ist eine Checkliste, die die engen Fragen der Reihe nach stellt.

Zentrale Erkenntnisse

- Eine schreibgeschützte Session ist der richtige Standard für Untersuchungen, und sie funktioniert nur, wenn die häufige Aufgabe keine Genehmigung braucht.
- Schreiben ist eine andere Handlung als Lesen, deshalb läuft es über einen eigenen Antrag, eine eigene Prüfung, eine eigene Zeitbegrenzung und eine eigene Aufzeichnung. Der Antrag trägt eine `fix_id` und einen `approved_hash` aus dem Merge-Eintrag, seine Tripel werden aus der Datenkorrektur abgeleitet, und der Genehmiger sieht die Anweisung.
- Die sichere Reihenfolge einer Datenkorrektur ist Zählung, Snapshot, Transaktion, Schutzprüfung der Zeilenzahl, Nachbedingung, Commit, und der Runner, nicht das Gedächtnis der Person, sollte sie durchsetzen, wobei das Gateway die Schreibvorgänge der Session nur vom Runner weiterleitet.
- Eine Schutzprüfung macht eine Klasse von Fehlern laut und eine Datenkorrektur nicht richtig, weshalb Prüfung, schmale Grants und ein Snapshot vor der Änderung als getrennte Schichten existieren.
- Eine Änderung an einer Restricted-Spalte ist keine Datenkorrektur: Sie läuft über `admin-console` zu `payments-service` mit einem zweiten Genehmiger und einem Eintrag in `audit_events`, und Break-Glass ist kein Weg daran vorbei.
- Eine wiederholte manuelle Datenkorrektur ist ein fehlendes Produktfeature, und jede, die zu einer Aktion in `admin-console` wird, ist ein wiederkehrender Schreibvorgang, der keine Person mehr auf dem Primary braucht.
- Erreicht eine Anweisung jede Zeile, lautet die nützliche Frage, was am Design es erlaubt hat, und die Antwort ist eine Liste fehlender Schutzprüfungen, keine Person.

Datenkorrekturen ändern, was in den Tabellen steht; die nächste Art von Änderung verändert die Tabellen selbst. Kapitel 11 wendet sich den Schemaänderungen zu, der Frage, wer sie mit welcher Identität ausführt, und

der Gewohnheit, sie abwärtskompatibel zu halten, damit die Reparatur eines Tages nicht der Ausfall des nächsten wird.

Kapitel 11 — Schemaänderungen und Zugriff für Migrationen

Stellen Sie sich einen Donnerstagnachmittag bei Ledgerly vor. Das Rechnungsteam möchte in `invoices` eine neue Spalte, damit die Steuer getrennt vom Gesamtbetrag ausgewiesen werden kann, und die Änderung ist klein: eine Spalte, ein Default, ein Backfill für alte Zeilen. Robin, diese Woche in der Rufbereitschaft, prüft den Pull Request und genehmigt ihn. Die Pipeline führt die Migration als `ledgerly_migrator` aus, das Deployment wird fertig, und etwa neunzig Sekunden lang hängt jede Anfrage an `billing-api`, die eine Rechnung berührt, und schlägt dann fehl.

An der Migration war nichts falsch, und niemand hat etwas Ungewöhnliches getan. Eine lange Batch-Transaktion von `invoice-worker`, die offen blieb, während sie auf einen externen Mail-Service wartete, hielt eine Sperre auf der Tabelle, die Migration reihte sich dahinter ein, und jede Anweisung, die nach der Migration eintraf, reihte sich hinter der Migration ein. Alex vom Platform-Team findet die Ursache schnell. Dann kommt die schwierigere Frage: Die Migration lief mit der Macht des Eigentümers jeder Tabelle, sie lief unbeaufsichtigt, und zwischen dieser Macht und dem Hauptbuch stand nur die Prüfung eines Diffs. Wie sollten Schemaänderungen den Primary erreichen, mit welcher Identität, und wie verhindern wir, dass sie entweder zu einer Lücke im Zugriffsmodell oder zu einem Ausfall werden?

Dieses Kapitel fügt dem Zielzustand einen Schritt hinzu: Migrationen laufen als `ledgerly_migrator` aus der Pipeline, und nichts sonst hat die Macht, das Schema zu ändern.

Im Modell mit fünf Teilen aus Kapitel 1 ist eine Migration die **Ausführung** mit der weitesten Begrenzung, die es gibt. Die Pipeline hält die Privilegien des Eigentümers von Entwurf her, das Risiko liegt also nicht darin, wer ein Credential hat. Es liegt darin, wozu die Migration berechtigt ist, wie ihre Anweisungen ausgeführt werden und welche Aufzeichnung bleibt. Für Mi-

grationen bedeutet das eine kurze Liste von Anweisungsklassen, die auf jeder Lane erlaubt sind (Autorisierung), einen Pfad, der entscheidet, wann und wie eine Anweisung läuft, mit kurzer Sperrwartezeit und einem Retry (Ausführung), und eine Aufzeichnung, die der Eigentümer nicht umschreiben kann (Zurechenbarkeit).

Das Problem und warum es schwierig ist

Eine Schemaänderung ist das Privilegierteste, was die meisten Teams an einem gewöhnlichen Tag mit einer Datenbank tun. Das Eigentum erlaubt dem Eigentümer, eine Tabelle zu löschen, den Typ einer Spalte zu ändern, Grants umzuschreiben und, bei vielen Engines, die Einschränkungen zu umgehen, die für alle anderen gelten. Kapitel 3 hat diese Macht `ledgerly_owner` gegeben, einer Rolle ohne Login, und `ledgerly_migrator` darf sie nur annehmen, solange eine Migration läuft. Diese Regelung beantwortet, wem das Schema gehört. Sie beantwortet nicht, wer eine Migration starten darf, was eine laufende Migration tun darf oder was mit allen anderen geschieht, während sie läuft.

In dieser einen Tätigkeit stecken drei getrennte Risiken.

Das erste ist die **Befugnis**. Wer den Migrationsschritt der Pipeline auslösen kann, kann praktisch Anweisungen auf Eigentümerebene in der Produktion ausführen. Dauerhaft ist die Befugnis, im Sinne von Kapitel 3: eine dauerhafte Befugnis des Eigentümers, ausgeübt über ein kurzlebiges Credential. Die Grenzen dafür sind verfahrensmäßig: Das Credential lebt nur in der Pipeline, es läuft mit dem Job ab, und die Pipeline ist der einzige zugelassene Weg, es zu verwenden. Diese Grenzen sind nur so gut wie die Zugriffskontrollen der Pipeline selbst, wodurch die Pipeline Teil der Sicherheitsgrenze der Datenbank wird, ob das jemand so eingezeichnet hat oder nicht.

Das zweite ist die **Verfügbarkeit**. Viele Schemaänderungen brauchen Sperren, und an der Sperre entsteht der Schaden. Eine Anweisung, die auf eine Sperre warten muss, wartet nicht nur: Bei vielen Engines reihen sich auch die Anweisungen ein, die hinter ihr eintreffen. Eine Migration, die ihre Sperre nicht in der ersten Sekunde bekommt, kann die Tabelle für alle schließen, solange die älteste Transaktion vor ihr offen bleibt. Die Größe

der Änderung spielt kaum eine Rolle. Eine Änderung an einer einzigen Spalte kann so störend sein wie ein Rewrite, wenn die falsche Transaktion im Weg ist.

Das dritte ist **Drift**. Ein Schema, das durch Migrationen definiert und zusätzlich von Hand bearbeitet wird, ist ein Schema, das niemand vollständig beschreiben kann. Ein Hotfix-Index, in einer schlechten Nacht von einer Konsole aus angelegt, ist nützlich, und ein Jahr später ist er ein Unterschied zwischen Umgebungen, an den sich niemand erinnert, samt Grants, die zu keiner Datei passen.

Jedes davon lässt sich durch Entwurf verringern. Eine Migrationsrolle, die von der Pipeline begrenzt wird, eine Sperrwartezeit, die schnell fehlschlägt, eine Form der Änderung, die nie eine lange Sperre braucht, und eine Prüfung, die das laufende Schema mit dem erwarteten vergleicht, lassen sich alle bauen. Eine Richtlinie, die sagt „Seien Sie vorsichtig mit Migrationen“, gehört nicht dazu.

Wer DDL ausführt, und als welche Rolle

DDL (Data Definition Language) ist die Familie von Anweisungen, die Schemaobjekte anlegen, ändern und löschen, im Gegensatz zu den Anweisungen, die Zeilen lesen und schreiben. Die Regel dieses Kapitels ist kurz: DDL läuft als `ledgerly_migrator`, aus der Pipeline, und keine Anwendungsrolle kann es ausführen.

Die Anwendungsrollen halten kein DDL

Keine der Rollen `app_*` aus Kapitel 3 besitzt eine Tabelle oder darf eine anlegen, ändern oder löschen. Das ist leicht gesagt und ein wenig kostspielig durchzuhalten, denn viele Frameworks können beim Start Tabellen anlegen oder ändern, und manche Teams schalten das ein, weil es einen Schritt aus dem Deployment nimmt. Der Preis dieser Bequemlichkeit ist, dass der Service dann Eigentum braucht, und ein kompromittierter oder verwirrter Service kann löschen und neu vergeben. Ich würde die automatische Schemaverwaltung in der Produktion abschalten und für Entwicklungsdatenbanken behalten, in denen niemand eine gelöschte Tabelle erklären muss.

Die sichtbare Nebenwirkung ist, dass ein Deployment mit einer Schemaänderung jetzt aus zwei Schritten besteht: der Migration und dann dem Rollout des Services. Die Reihenfolge zwischen beiden ist eine Entwurfsfrage, die das unten beschriebene Expand-and-Contract-Pattern beantwortet.

Der Migrator und der Eigentümer

`ledgerly_migrator` hat einen Login, den die Pipeline verwendet; er ist Mitglied von `ledgerly_owner` und hält selbst keine Datenprivilegien. Die Mitgliedschaft gibt dem Migrator die Privilegien des Eigentümers, macht ihn aber nicht zum Eigentümer. Bei vielen Engines gehört ein neues Objekt der Rolle, die beim Anlegen aktiv ist, sodass eine Migration, die unter dem eigenen Login des Migrators läuft, ihre Tabellen im Eigentum des Migrators hinterließe. Die Regel ist deshalb bei jeder Engine dieselbe: Eine Migration wechselt immer ausdrücklich zu `ledgerly_owner`, weil das Eigentum der aktiven Rolle folgt und nicht den geerbten Privilegien. Jedes Objekt, das sie anlegt, gehört dann `ledgerly_owner`, und die Migration kehrt am Ende zur eigenen Rolle des Migrators zurück. In einer Notation, die zu keiner Engine gehört:

```
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 })
```

Vier Details sind wichtig. Das Credential wird diesem Job zur Laufzeit von der Workload Identity der Pipeline ausgestellt (das Pattern aus Kapitel 4) und läuft mit dem Job ab, sodass eine kopierte Pipeline-Variable monatelang kein funktionierendes Eigentümer-Credential halten kann und nichts im Repository eines hält. Der Wechsel zur Eigentümerrolle ist auf die Anweisungen einer einzigen Migration begrenzt. Die Aufzeichnung, dass eine Migration angewendet wurde, wird im selben Lauf geschrieben, sodass die Datenbank ihre eigene Version kennt, von der die Drift-Prüfung weiter unten abhängt. Und der externe Eintrag wird vor der ersten Anweisung des Eigentümers geschrieben, weil `recordApplied` in einer Datenbank liegt, die der Eigentümer umschreiben kann. Der Preis ist, dass ein nicht verfügbarer Audit-Speicher die Migration stoppt, bevor sie beginnt. Das ist die umgekehrte Reihenfolge wie bei der Datenkorrektur in Kapitel 10, wo das Session-Log des Gateways die Anweisungen bereits aufzeichnet: Hier sieht außer der Pipeline niemand, was die Session des Eigentümers tut, deshalb

kommt die Aufzeichnung zuerst. Jeder Start-Eintrag bekommt einen passenden Abschluss, `migration-failed` oder `migration-applied`, und `isApplied` macht einen Retry auch dann sicher, wenn das abschließende Anhängen an das Audit-Log der fehlgeschlagene Schritt war.

Bei manchen Engines können das Festhalten der Version und das Ausführen des DDL in einer Transaktion liegen, bei anderen beendet eine DDL-Anweisung die Transaktion um sie herum. Wissen Sie, welche Art Sie haben. Kann ein Fehler das Schema geändert und die Version unprotokolliert hinterlassen, muss die Migration so geschrieben sein, dass ein zweiter Lauf harmlos ist.

Menschen halten den Migrator nicht

Menschen verbinden sich nie als `ledgerly_migrator`. Braucht eine Person eine Schemaänderung, schreibt sie eine Migration, und die Pipeline führt sie aus. Das klingt nach Reibung, und das ist es auch, aber es bringt eine nützliche Eigenschaft: Jede Nutzung der Macht des Eigentümers entspricht einem Pipeline-Lauf, der einen Commit, eine Prüfung und ein Log hat. Liegt das Migrator-Credential außerdem in einem Passwortmanager des Teams, verschwindet diese Eigenschaft, und mit ihr der Grund, überhaupt einen Migrator zu haben.

Was die Pipeline erreichen kann

Der Pipeline das Migrator-Credential zu geben verlagert eine große Macht in ein System, das viele Menschen beeinflussen können. Es lohnt sich, das ausdrücklich zu sagen, denn hier kann dieser Entwurf unbemerkt schiefgehen.

Was die Pipeline tun kann

Ein Job, der `runMigration` ausführt, kann jedes Objekt im Schema anlegen, ändern oder löschen, jeden Grant auf Objekten ändern, die dem Eigentümer gehören, und die beiden Append-only-Tabellen `ledger_entries` und `audit_events` ändern. Keine Anwendungsrolle und keine menschliche Rolle kann Zeilen in diesen Tabellen bearbeiten, auch `human_breakglass` nicht; die eine strukturelle Ausnahme ist, dass der Eigentümer, und damit eine laufende Migration, sie ändern kann. Die Kontrolle für diese Ausnahme ist kein Privileg. Sie ist die Prüfung.

Die Frage an die Pipeline lautet also: Wer kann sie dazu bringen, etwas auszuführen? Das umfasst Personen, die Änderungen per Merge in den Deployment-Branch übernehmen, Personen, die die Pipeline-Definition bearbeiten können, Personen, die einem Job, der das Credential hält, einen Schritt hinzufügen können, und jeden, der die Umgebung des Jobs während des Laufs lesen kann.

Die Reichweite begrenzen

Ich würde diese Reichweite auf fünf Arten begrenzen, grob danach geordnet, wie viel jede davon wegnimmt.

1. **Nur ein Job hält das Credential.** Der Migrations-Job für die Produktionsumgebung ist der einzige Ort, an dem der Migrator-Login ausgestellt werden kann, und er bekommt ihn durch die Identität des Jobs, nicht durch eine Variable, die jeder Job lesen kann. Builds ungeprüfter Branches, Test-Jobs und Pull-Request-Pipelines erhalten es nie.
2. **Die Pipeline-Definition wird wie die Migrationen geprüft.** Eine Änderung am Migrations-Job, an seinen Berechtigungen oder an den Branch-Regeln, die ihn auslösen, ist eine Änderung der Reichweite des Eigentümers und braucht denselben Reviewer, der auch eine Migration prüfen würde.
3. **Der Auslöser ist ein geschützter Branch.** Der Merge dorthin erfordert eine Prüfung durch eine Person, die nicht der Autor ist, und das ist die Vier-Augen-Kontrolle von Ledgerly für Schemaänderungen. Bei Migrationen, die Grants, Eigentum oder die Append-only-Tabellen berühren, muss der Reviewer aus dem Platform-Team kommen.
4. **Das Credential ist kurzlebig.** Ein Job, der fehlschlägt und ein Log mit einem Secret hinterlässt, hinterlässt ein Secret, das bereits abgelaufen ist.
5. **Jeder Lauf wird aufgezeichnet.** Die Pipeline schreibt, was sie angewendet hat, als welcher Login, aus welchem Commit, in einen Speicher, den die Pipeline nicht umschreiben kann, und sie schreibt den Start-Eintrag, bevor die Migration läuft. Kapitel 14 behandelt, wo dieser Speicher liegt.

Die Prüfung selbst kann teilweise mechanisch sein. Eine kleine Kontrolle kann die Migration vor einem Menschen lesen und die Anweisungen markieren, die einen genaueren Blick brauchen:

```
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
```

In der normalen Lane entscheidet die Kontrolle nichts. Sie lenkt die Aufmerksamkeit des Reviewers auf die Zeilen, die den Zugriff ändern, eine Sperre halten oder Daten zerstören, und sie macht das Fehlen eines Grants sichtbar. Eine Kontrolle, die einen Merge bei jeder Markierung blockiert und keine Möglichkeit bietet, eine zu bestätigen, wird innerhalb eines Monats umgangen, deshalb verlangt eine Markierung einen benannten Genehmiger und verbietet die Änderung nicht. Die beschleunigte Lane ist der eine Ort, an dem die Kontrolle ablehnt: Eine nicht leere Liste `refusals` lässt den Job scheitern, bevor er läuft, und kein Genehmiger kann das aufheben. Der nächste Abschnitt erklärt, warum.

Die normale Lane lehnt nichts grundsätzlich ab, und das ist eine Entscheidung: Eine harte Liste verbotener Anweisungen wäre eine zweite Richtlinie neben der Deny-, Gate- und Allow-Matrix aus Kapitel 9. Das Löschen von `ledger_entries` wird doppelt markiert, als Append-only-Änderung und als zerstörerische, und das Platform-Team muss dazu Ja sagen.

Warum nicht eine Person mit einem Ticket

Manche Teams ziehen es vor, dass ein DBA jede Migration nach der Genehmigung von Hand anwendet. Bei einer kleinen Datenbank mit ein oder zwei Schemaänderungen im Monat kann das eine vernünftige Wahl sein, und die Frage nach der Reichweite der Pipeline entfällt. Der Preis zeigt sich, wenn die Person abwesend ist, wenn die Änderung auf mehreren Umgebungen identisch angewendet werden muss oder wenn der manuelle Schritt von dem abweicht, was geprüft wurde. Die Schwäche der Pipeline ist ihre Reichweite; die Schwäche des manuellen Wegs ist, dass nichts erzwingt, dass das Angewendete dem Genehmigten entspricht. Ich neige zur Pipeline mit den oben genannten Grenzen, und ich würde die Wahl überdenken, wenn sich die Zugriffsliste der Pipeline selbst als nicht kurz haltbar erwiese.

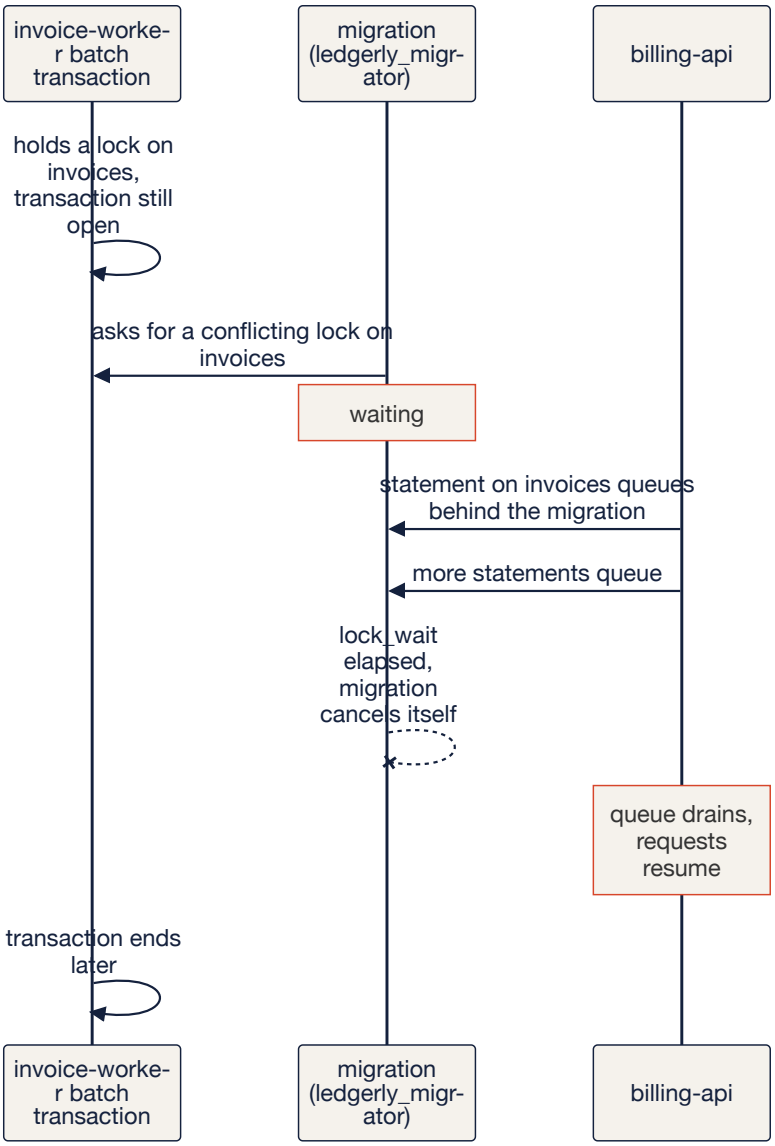
Sperren und das Verfügbarkeitsrisiko

Zurück zu den neunzig Sekunden bei Ledgerly. Der Mechanismus dahinter ist bei den meisten Engines derselbe, und die Verteidigung besteht aus einer Handvoll Gewohnheiten.

Wie eine wartende Migration eine Tabelle schließt

Eine Migration, die `invoices` ändert, braucht eine Sperre, die mit gewöhnlichen Lese- und Schreibvorgängen kollidiert. Sie kann diese Sperre nur nehmen, wenn keine Transaktion mehr offen ist, die eine kollidierende Sperre hält. Ist eine lange Transaktion offen, wartet die Migration. Während sie wartet, reihen sich neue Anweisungen auf der Tabelle hinter der anstehenden Anforderung der Migration ein, weil sie vorbeiziehen zu lassen sie aushungern würde. Die Tabelle ist dann faktisch geschlossen, bis die alte Transaktion endet oder die Migration aufgibt.

Die Sequenz unten zeigt die Reihenfolge der Ereignisse. Achten Sie auf den Punkt, an dem die Sperrwartezeit der Migration einen Ausweg gibt, bevor die Warteschlange wächst.



Der Punkt ist der Selbstabbruch. Ohne Sperrwartezeit wartet die Migration so lange, wie diese Batch-Transaktion offen bleibt, und `billing-api` steht die ganze Zeit in der Warteschlange. Mit kurzer Sperrwartezeit scheitert die Migration nach wenigen Sekunden, die Warteschlange leert sich, und die Pipeline kann es später erneut versuchen.

Eine kurze Sperrwartezeit und ein geduldiger Retry

Kapitel 5 hat für `ledgerly_migrator` `lock_wait 3s` und keinen Statement-Timeout festgelegt. Diese Paarung ist Absicht. Eine Schemaänderung darf lange brauchen, um ihre Arbeit zu tun, aber sie sollte sehr kurz warten, um anzufangen. Der Retry gehört dann der 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
```

Dazu einige Anmerkungen. Eine fehlgeschlagene Sperrwartezeit ist nur dann ein sauberes No-op, wenn noch kein Commit ausgeführt wurde, weshalb eine Migration eine Markierung `retrySafe` trägt: Sie ist eine Transaktionseinheit, oder ihr sperrender Schritt kommt zuerst. Eine mehrstufige Migration bei einer Engine, auf der jede DDL-Anweisung für sich einen Commit ausführt, wird nicht blind wiederholt, weil der zweite Versuch auf die Arbeit des ersten träfe. Sie stoppt, und eine Person entscheidet. Die Pause wächst mit der Zahl der Versuche, damit wiederholte Versuche nicht selbst zu einem Lastmuster werden. Und der Endzustand, `MigrationBlocked`, ist eine Frage an einen Menschen: herausfinden, welche Transaktion im Weg ist, und entscheiden, ob man wartet, sie abbricht oder neu terminiert. Diese Entscheidung bedeutet, die Session von jemandem zu beenden, was eine eigene Zugriffsfrage ist und in Kapitel 15 gehört, wenn es während eines Incidents geschieht.

Lange Anweisungen und Online-Änderungen

Manche Operationen halten eine Sperre, solange sie laufen, etwa das Neuschreiben einer großen Tabelle, um den Typ einer Spalte zu ändern. Bei ihnen ist nicht die Sperrwartezeit das Problem, sondern die Dauer. Engines unterscheiden sich darin, welche Anweisungen schnell sind, und die Antwort ändert sich mit der Version der Engine, also prüfen Sie jede einzeln. Werkzeuge für Online-Schemaänderungen und nicht blockierende Index-Builds vermeiden die lange Sperre, aber sie legen Objekte an und löschen sie, brauchen also dieselbe Eigentümer-Reichweite wie jede Migration und laufen als Migrator innerhalb der Pipeline. `classifyMigration` markiert diese Anweisungen als solche mit lang gehaltener Sperre, und eine Migration mit lang gehaltener Sperre bekommt ein geplantes Zeitfenster und eine benannte Person, die zuschaut, festgehalten in der Beschreibung der Migration. Das ist kein Versagen der Automatisierung; manche Änderungen sind Ereignisse.

Expand and Contract

Die sicherste Schemaänderung ist eine, mit der alter und neuer Anwendungscode leben können. Das Pattern dafür heißt **Expand and Contract**: Das Schema in additiven Schritten ändern, die Anwendung hinüberziehen und erst dann entfernen, was nicht mehr benutzt wird. Für dieses Buch ist es wichtig, weil jeder Schritt eine kleine Migration mit einer kleinen Sperre ist und weil es Deployments umkehrbar hält.

Nehmen Sie eine hypothetische Änderung bei Ledgerly, die nur die Form zeigen soll: `total_minor` in `invoices` in `amount_minor` umzubenennen. Das kanonische Schema behält `total_minor`, und das Beispiel nimmt eine Variante an, in der die Spalte umbenannt wird. Die Schritte sehen so aus:

1. **Expand.** Eine Migration fügt `amount_minor` als nullable Spalte hinzu, mit einer kurzen Sperre. Alter Code ignoriert sie.
2. **Beides schreiben.** Neue Releases von `billing-api` und `invoice-worker` schreiben die neue Spalte ebenso wie die alte, sodass alte und neue Releases nebeneinander laufen können.
3. **Backfill.** Alte Zeilen bekommen Werte für die neue Spalte, wie unten beschrieben.

4. **Die neue Spalte lesen.** Verhält sich das Release schlecht, ist die alte Spalte weiterhin korrekt, der Rollback ist also ein Deployment und keine Reparatur.
5. **Contract.** Nach einer Zeit, in der nichts `total_minor` liest oder schreibt, löscht eine abschließende Migration sie. Das ist der einzige zerstörerische Schritt, und er hat einen eigenen Kontrollpunkt: als zerstörerisch geprüft und nie in dem Release, das die neue Spalte eingeführt hat.

Expand and Contract kostet Zeit und Aufmerksamkeit. Eine Änderung, die eine Migration sein könnte, wird zu vier oder fünf, verteilt auf mehrere Deployments, und in den mittleren Schritten trägt die Anwendung Code für beide Formen. Für ein kleines Team mit einem Service und einem Wartungsfenster kann eine einzige Migration die bessere Wahl sein. Für Ledgerly, wo fünf Services dieselben Tabellen lesen und es kein Fenster gibt, das das Geschäft akzeptiert, hält das Pattern die Sperre kurz und den Rollback billig.

Backfills in der Migration, Datenkorrekturen außerhalb

Schritt 3 ist eine Massenänderung an Daten, und dieselbe Regel würde für eine Aufteilung des Gesamtbetrags in Netto und Steuer gelten, mit einem Unterschied: Die alte Tabelle enthält keine Steuerangabe, sodass dieser Backfill nicht berechnet werden könnte und die neuen Spalten leer lassen müsste, bis eine Datenkorrektur sie füllt. Kapitel 10 definiert eine Datenkorrektur als eine Änderung an Produktionsdaten, die eine Person außerhalb des normalen Anwendungspfads vornimmt, um einen Zustand zu reparieren, den die Anwendung nicht reparieren kann; das Buch braucht hier also eine Linie: Ein Backfill, der zu einer Schemaänderung gehört, läuft innerhalb der Migration als `ledgerly_migrator`; jede andere Massenänderung an Daten ist eine Datenkorrektur und folgt Kapitel 10.

Was macht einen Backfill zu einem Teil der Schemaänderung?

- Sein einziger Zweck ist, der neuen Struktur ihre ersten Werte zu geben, nach einer Regel, die in der Migration steht.
- Er ist in der Migration beschrieben, wird mit ihr geprüft und von der Pipeline angewendet.

- Jeder Wert, den er schreibt, ließe sich aus Daten neu berechnen, die bereits in der Tabelle stehen.

Trifft eines davon nicht zu, ist es eine Datenkorrektur. Die Umbenennung zeigt, wo die Linie verläuft: Jeder Wert, der in `amount_minor` geschrieben wird, ist `total_minor` derselben Zeile, eine Regel, die die Migration nennt und die ein Reviewer prüfen kann. Angenommen, das Team bemerkt beim Backfill, dass manche alten Rechnungen ein falsches `total_minor` haben. Diese Beträge zu korrigieren ist nicht Teil der Schemaänderung. Es folgt Kapitel 10, mit einer Zählung der betroffenen Zeilen, einer Schutzprüfung der erwarteten Zeilenzahl und einem zweiten Reviewer, denn eine Kopie, die einen falschen Wert bewahrt, ist trotzdem ein korrekter Backfill. Die Macht des Migrators ist groß, und er sollte nicht der Weg für jede unbequeme Änderung werden.

Bei einer großen Tabelle läuft der Backfill in Batches, jeder eine kurze Transaktion über eine begrenzte Menge von Zeilen. Er pausiert, wenn die Replikationsverzögerung wächst (Kapitel 6), gibt sichtbar auf, wenn sich die Verzögerung nicht löst, und lässt sich stoppen und neu starten, ohne Arbeit zu verlieren.

Dieselbe Regel gilt für die beiden Append-only-Tabellen. Eine Migration darf `ledger_entries` eine Spalte hinzufügen, aber sie schreibt bestehende Einträge nicht um, um diese Spalte mit korrigierten Werten zu füllen; eine Korrektur ist ein neuer Eintrag, ausgestellt über den Korrekturpfad von `payments-service`. Eine Migration, die diese Tabellen doch berührt, wird vom Klassifizierungsschritt oben markiert und geht zur Prüfung an das Platform-Team.

Hotfix-DDL und Drift

Jedes Team erlebt irgendwann die Nacht, in der ein Index eine langsame Abfrage beheben würde und die Pipeline, sagen wir, zwanzig Minuten braucht. Die Frage ist, was das Zugriffsmodell zur Abkürzung sagt.

Es gibt keine Abkürzung über die Konsole, also bauen Sie eine schnelle Lane

Im Zielzustand hält niemand den Login des Eigentümers. `human_breakglass` schreibt `INSERT` und `UPDATE` und trägt kein DDL, und der Superuser ist ein versiegeltes Instrument, das nicht für die tägliche Arbeit benutzt wird (Kapitel 3). Eine Person auf dem Primary hat also keinen legitimen Weg, einen Index von Hand anzulegen. Das ist Absicht, und es verlagert die Frage auf die Pipeline: Wie schnell kann sie eine geprüfte Änderung ausführen, wenn der Bedarf dringend ist?

Ich würde der Pipeline eine **beschleunigte Lane** geben statt einer Umgehung. Die Lane nutzt denselben Job, dasselbe Credential und dieselbe Aufzeichnung; sie unterscheidet sich im Zeitablauf, nicht darin, wer genehmigen muss. Das Vier-Augen-Prinzip bleibt: Ein Reviewer, der nicht der Autor ist, genehmigt weiterhin, und der Autor merged nicht allein. Gelockert wird die geplante Wartezeit und, für eine kurze Allowlist von Anweisungsklassen, die Anforderung der Platform-Team-Prüfung, mit einer nachgelagerten Prüfung durch das Platform-Team am nächsten Werktag. `classifyMigration` setzt die Allowlist durch: Einen Index zu bauen, einen Index zu löschen und eine nullable Spalte hinzuzufügen sind zulässig. Eine Anweisung, die Grants, Eigentum oder eine View ändert, eine Append-only-Tabelle berührt oder Daten zerstört, ist es nicht, denn das sind die Änderungen, für die es das Platform-Review gibt, und eine Migration, die eine davon enthält, wird in der Lane abgelehnt und läuft über die normale. Kapitel 15 beschreibt den Zugriff bei Incidents; der Punkt hier ist, dass Dringlichkeit die Geschwindigkeit des Prozesses ändert, nicht die Identität, die das DDL ausführt, und nicht die Prüfung, die eine gefährliche Anweisung bekommt.

Der Preis ist, dass die beschleunigte Lane eine dauerhafte Option ist, manche Vorsicht zu überspringen, und eine solche Option wird genutzt. Verfolgen Sie, wie oft sie verwendet wird, und werten Sie eine steigende Zahl als Zeichen dafür, dass die normale Lane zu langsam ist, nicht dafür, dass das Team nachlässiger geworden ist.

Drift

Drift ist jeder Unterschied zwischen dem Schema, das die Migrationen beschreiben, und dem Schema, das in der Produktion existiert. Er entsteht durch einen von Hand angewendeten Hotfix, durch eine Migration, die nach ihrer Anwendung bearbeitet wurde, durch eine Änderung auf Eigentümerebene durch ein Werkzeug, an das sich niemand erinnerte, und durch Grants, die außerhalb der Zugriffsdefinition geändert wurden. Drift im Schema überrascht die nächste Migration. Drift bei Grants ist eine Zugriffsänderung, die keine Prüfung gesehen hat.

Eine geplante Prüfung kann beide Arten finden:

```
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")
```

Zwei Dinge machen das nützlich. Der erwartete Zustand wird aus den Migrationen und der Zugriffsdefinition aus Kapitel 3 abgeleitet, es gibt also genau eine Quelle der Wahrheit zum Vergleich. Und die Prüfung deckt Grants ab, nicht nur Tabellen, denn ein Grant, der außerhalb jeder Datei auftaucht, ist für dieses Buch die wichtigere Art von Drift. Wird ein Unterschied gefunden, gibt es zwei Auflösungen: eine Migration schreiben, die die Änderung festhält, wenn sie bleiben soll, oder eine, die sie rückgängig macht, wenn nicht. Drift aufzulösen, indem man die Aufzeichnung dessen bearbeitet, was angewendet wurde, versteckt die Änderung, statt sie zu prüfen.

> **Hinweis:** „Beheben“ Sie Drift nicht, indem Sie eine korrigierende Anweisung direkt als Superuser oder Eigentümer anwenden. Die Anweisung, die den Unterschied beseitigt, ist selbst eine ungeprüfte Änderung an der Produktion. Legen Sie sie in eine Migration und lassen Sie sie durch die Lane laufen.

Neue Tabellen, neue Spalten und die Grants, die folgen

Kapitel 3 hat für neue Tabellen einen leeren Default gewählt: Eine Tabelle ist für niemanden außer dem Eigentümer lesbar, bis eine Migration einen Grant nennt. Dieser Abschnitt ist die andere Hälfte dieser Entscheidung, denn in Migrationen hält die Entscheidung oder wird vergessen.

Grants leben in der Migration

Jede Migration, die eine Tabelle anlegt, nennt ihre Grants in derselben Datei. Ein Reviewer sieht dann die gesamte Zugriffswirkung einer Änderung in einem Diff, und das ist das Argument für den leeren Default. Für eine neue Tabelle, `invoice_adjustments` (eine illustrative Tabelle, nicht Teil des kanonischen Schemas), sieht die Datei so aus. Ihre Freitextspalte `reason` ist für den ungünstigsten Fall als Confidential klassifiziert, so wie `invoice_lines.description`, sodass andere Leser als der Eigentümer der Tabelle Grants auf Spaltenebene erhalten, die sie auslassen:

```
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;
```

Einen Grant zu vergessen hat zwei mögliche Ausgänge, und der Unterschied ist der Grund für den leeren Default. Bei einem leeren Default scheitert der Service, der die Tabelle braucht, an seiner ersten Abfrage. Das ist laut, wird in einem Test oder beim ersten Deployment gefunden und durch eine einzeilige Migration behoben. Bei einem breiten Default-Privileg ist ein vergessener Grant in der anderen Richtung still: Die nächste Tabelle ist für alle lesbar, die der Default nennt, und niemand bemerkt es, weil nichts scheitert. Ein lautes Scheitern ist billiger als eine stille Offenlegung.

Das Scheitern ist nur laut, wenn etwas die Abfrage ausführt. Deshalb würde ich in die Pipeline einen Smoke-Schritt nach der Migration aufnehmen, der prüft, dass jede Anwendungsrolle die Tabellen nutzen kann, die sie nutzen soll. Die Anwendungsrollen haben keinen Login, der Schritt braucht also einen Mechanismus. Ich würde einen Test-Login verwenden, der Mitglied jeder Anwendungsrolle ist und nacheinander zu jeder wechselt, wo die Engine das unterstützt; die Alternative ist ein Test-Login pro Rolle. Beide Wege erweitern die Reichweite der Pipeline, denn ein Login, der jede Service-Rolle werden kann, ist für einen Angreifer mehr wert als jedes einzelne Service-Credential. Es wird deshalb von der Workload Identity des Jobs ausgestellt, läuft mit dem Job ab und wird nur für diesen Schritt verwendet. Dafür fängt der Schritt den vergessenen Grant ab, bevor es der Rollout des Services tut.

Neue Spalten in Tabellen, die bereits Grants haben

Eine neue Spalte verhält sich unterschiedlich, je nachdem, wie ihre Tabelle vergeben ist.

Grant-Stil der Tabelle	Eine neue Spalte ist …	Folge
Grant auf Tabellenebene	für jede Rolle lesbar, die den Grant hält	Eine neue sensible Spalte ist sofort offengelegt
Grant auf Spaltenebene	für niemanden lesbar, bis sie genannt wird	Eine neue Spalte muss vergeben werden, um genutzt zu werden

Bei Tabellen mit sensiblen Spalten, etwa `customers`, `bank_accounts` und `payment_tokens`, spricht das für Grants auf Spaltenebene, wo die Engine sie unterstützt, und das ist die Wahl, die Kapitel 3 getroffen hat. Die neue

Spalte beginnt dann geschlossen, und in der Migration, die sie hinzufügt, entscheidet das Team, welche Rollen sie lesen dürfen und ob sie Confidential oder Restricted ist (Kapitel 12 behandelt die Klassifizierung). Eine Migration, die einer dieser Tabellen eine Spalte ohne jede Grant-Anweisung hinzufügt, ist Anlass für eine Frage, und die Klassifizierungsprüfung kann sie stellen.

Wo die Engine keine Grants auf Spaltenebene hat, gilt der Ausweg aus Kapitel 3: Restricted-Spalten liegen in eigenen Tabellen wie `bank_account_secrets` und `payment_token_values`. Eine neue Restricted-Spalte gehört in eine dieser Tabellen, nie in die Haupttabelle, denn die Grants der Haupttabelle sind auf Tabellenebene, und die Obergrenze für JIT-Zugriff ist um sie gezogen. Das lässt sich leicht versehentlich verletzen, da es der naheliegende Schritt ist, die Tabelle zu erweitern, die alle schon kennen, und es ist die Art von Fehler, die nur ein Reviewer bemerkt, der den Ausweg kennt.

Views über sensible Tabellen

Eine View läuft bei vielen Engines mit den Rechten ihres Eigentümers, was bedeutet, dass eine View über eine Tabelle mit Restricted-Spalten offenlegen kann, was die eigenen Rollen ihrer Leser nicht könnten. Deshalb markiert `classifyMigration` `CREATE VIEW` als Zugriffsänderung, deshalb läuft die Migration, die eine anlegt, über das Platform-Team, und deshalb wird eine View über eine sensible Tabelle als Zugriffsänderung beurteilt und nicht als Bequemlichkeit.

Abwägungen und Kosten

Die Pipeline ist ein wertvolles Ziel. Ein Credential, das DDL auf Eigentümerebene ausführen kann, gehalten von einem System, das viele Menschen bearbeiten können, ist eine Konzentration von Macht. Eine geprüfte Pipeline hindert niemanden mit Reichweite auf Eigentümerebene daran, etwas Ungeprüftes zu tun; sie macht dieses Etwas im Nachhinein sichtbar, was eine schwächere Kontrolle ist, als es klingt. Die Grenzen in diesem Kapitel verringern das Risiko, ohne es zu beseitigen. Ist die Zugriffsliste Ihrer Pipeline lang und gehört sie niemandem, beheben Sie das, bevor Sie sich auf irgendetwas vom Rest verlassen.

Prüfungen bringen Verzögerung und sind nur so gut wie der Reviewer. Eine Vier-Augen-Kontrolle bei Migrationen verlangsamt kleine Änderungen und kann zur Formalität werden, besonders wenn der Reviewer, sagen wir, zehn Migrationen pro Woche sieht und jede eine Spalte ist. Die Klassifizierungsprüfung hilft, indem sie auf die gefährlichen Zeilen zeigt. Sie ersetzt keine Aufmerksamkeit. Eine Prüfung, die immer innerhalb von Sekunden genehmigt wird, ist das Signal, auf das man achten muss, und Kapitel 8 führt das Argument vollständig aus.

Expand and Contract vervielfacht die Zahl der Änderungen. Aus einer logischen Änderung werden mehrere Deployments. Für ein Team mit einem Wartungsfenster, einem Service und einer kleinen Tabelle lohnt es sich vielleicht nicht. Das Pattern zahlt sich aus, wenn viele Dinge die Tabelle lesen und niemand sie für die Dauer einer Änderung anhalten kann.

Eine kurze Sperrwartezeit lässt Migrationen scheitern. Das ist das beabsichtigte Verhalten, und es ist ein Preis. Auf einer stark genutzten Tabelle braucht eine Migration vielleicht mehrere Versuche, und jemand muss entscheiden, was nach dem letzten geschieht. Setzen Sie die Wartezeit für Ihre Last zu kurz an, alarmiert die Pipeline das Platform-Team grundlos. Stimmen Sie sie darauf ab, wie Ihre eigenen langen Transaktionen aussehen, und halten Sie das im Runbook fest.

Drift-Prüfungen erzeugen Befunde, die jemandem gehören müssen. Eine Prüfung, die einen Kanal alarmiert, den niemand liest, findet nichts. Entscheiden Sie, wer jede Klasse von Befunden verantwortet, bevor Sie die Prüfung einschalten.

Wann man diesen Entwurf nicht einsetzt. Für eine Wegwerf-Datenbank, einen Prototyp oder ein Projekt eines einzelnen Entwicklers ohne Kundendaten ist ein Migrator mit einer Eigentümerrolle und einer geprüften Pipeline mehr, als die Lage verlangt. Führen Sie Migrationen vom Rechner eines Entwicklers aus, halten Sie die Produktionsdatenbank vom kritischen Pfad fern, und fügen Sie die Pipeline hinzu, sobald jemand außer Ihnen von den Daten abhängt. Der Entwurf in diesem Kapitel ist für den Punkt gedacht, an dem das Schema von Services gemeinsam genutzt wird und eine falsche Änderung Kunden etwas kostet.

Häufige Fehler

- **Den Service beim Start seine eigenen Tabellen anlegen oder ändern lassen.** Das nimmt einen Deployment-Schritt weg und gibt dem Service Eigentum, was die Macht zum Löschen und Neuvergeben einschließt. Halten Sie die automatische Schemaverwaltung in der Produktion aus und führen Sie Migrationen als eigenen Schritt mit dem Migrator aus.
- **Das Migrator-Credential „für Notfälle“ in einem Passwortmanager des Teams aufbewahren.** Es ist verlockend, weil die Pipeline manchmal im ungünstigsten Moment ausfällt. Es macht jeden Inhaber zum Eigentümer und beseitigt die Eigenschaft, dass jede Nutzung von Eigentum einem Pipeline-Lauf entspricht. Bauen Sie stattdessen die beschleunigte Lane.
- **Eine Migration ohne Sperrwartezeit ausführen.** Die Migration wartet so lange wie die älteste Transaktion vor ihr, und jede Anweisung hinter ihr wartet ebenfalls. Geben Sie dem Migrator eine kurze Sperrwartezeit und legen Sie den Retry in die Pipeline.
- **Eine Datenkorrektur in einem Backfill verstecken.** Es fühlt sich effizient an, schlechte Werte zu korrigieren, solange die Tabelle ohnehin offen ist. Die Prüfung der Migration war für eine Spaltendefinition gedacht, nicht für eine Änderung an Beträgen von Kunden. Ist die Änderung keine Erstbefüllung der neuen Struktur nach einer genannten Regel, ist sie eine Datenkorrektur und folgt Kapitel 10.
- **Die alte Spalte im selben Release löschen, das die neue einführt.** Das spart eine Migration und beseitigt den Rollback. Halten Sie den Contract-Schritt getrennt, verzögert und als zerstörerisch geprüft.
- **Sich auf Default-Privilegien verlassen, damit neue Tabellen ohne Grant-Zeile funktionieren.** Ein breiter Default macht aus einem vergessenen Grant eine stille Offenlegung. Halten Sie den Default leer und lassen Sie jede Migration ihre Grants nennen.
- **Eine Restricted-Spalte in einer Haupttabelle ergänzen, bei einer Engine ohne Grants auf Spaltenebene.** Die Grants der Tabelle sind auf Tabellenebene, und die JIT-Obergrenze ist um sie herum gebaut. Legen Sie Restricted-Spalten in die separaten Secret-Tabellen.
- **Einen Drift-Befund als Rauschen behandeln.** Ein Grant, der in der Produktion existiert und nirgends im Code, ist eine Zugriffsentschei-

dung, die niemand geprüft hat. Lösen Sie sie mit einer Migration auf, die ihn entweder festhält oder rückgängig macht.

- **Eine Migration nach ihrer Anwendung bearbeiten.** Die Umgebungen, die die alte Version ausgeführt haben, unterscheiden sich nun von denen, die die neue ausführen. Fügen Sie stattdessen eine neue Migration hinzu.

Zentrale Erkenntnisse

- DDL läuft als `ledgerly_migrator` aus der Pipeline, für die Dauer einer Migration als Eigentümer, und keine Anwendungsrolle hält ein DDL-Privileg.
- Der Migrator hält eine dauerhafte Befugnis des Eigentümers, ausgeübt über ein kurzlebiges Credential: Ein Job hält es, es wird per Workload Identity ausgestellt, jeder Lauf wird vor seinem Start aufgezeichnet, und Menschen verwenden es nie.
- Weil die Pipeline die Macht des Eigentümers erreichen kann, werden ihre eigene Zugriffsliste und Definition wie eine Migration geprüft; Migrationen, die Grants oder die Append-only-Tabellen berühren, gehen an das Platform-Team, und die beschleunigte Lane akzeptiert nur eine Allowlist von Anweisungsklassen, die `classifyMigration` durchsetzt.
- Eine kurze Sperrwartezeit mit einem Retry in der Pipeline verhindert, dass eine blockierte Migration eine Tabelle schließt, und Online-Werkzeuge und additive Schritte erledigen Änderungen, die sonst lange eine Sperre halten würden.
- Expand and Contract hält alten und neuen Code lauffähig und macht den Rollback zu einem Deployment, auf Kosten von mehr Migrationen; ein Backfill, der die neue Struktur nach einer genannten Regel füllt, läuft innerhalb der Migration, und jede andere Massenänderung an Daten ist eine Datenkorrektur und folgt Kapitel 10.
- Drift im Schema oder bei Grants wird gefunden, indem man den laufenden Zustand mit dem vergleicht, was Migrationen und Zugriffsdefinition beschreiben, und sie wird durch eine Migration aufgelöst, nicht durch eine stille Bearbeitung.
- Jede Migration nennt die Grants für das, was sie anlegt, und hält Defaults leer, sodass ein vergessener Grant laut scheitert, statt eine Tabel-

le still offenzulegen.

Schemaänderungen sind der eine Ort, an dem eine Pipeline und nicht eine Person die meiste Macht über die Daten hält. Der nächste Teil wendet sich von der Frage, wer die Datenbank erreichen kann, dem zu, was in ihr steckt: Kapitel 12 fragt, welche Spalten Schutz verdienen, wie man sie klassifiziert und wie man sie maskiert oder zurückhält, ohne die Arbeit zu stören, die sie braucht.

TEIL IV

Daten schützen

Kapitel 12 — Sensible Daten: Klassifizierung, Maskierung und Minimierung

Stellen Sie sich einen Montag bei Ledgerly vor. Ein Kunde schreibt dem Support, eine Auszahlung sei nicht angekommen, und fragt, ob das hinterlegte Bankkonto das richtige ist. Sam vom Support öffnet die Seite des Kunden in `admin-console` und sieht die vollständige Kontonummer auf dem Bildschirm. Sie im Chat zurückzulesen ist der schnellste Weg zu antworten, und ein Screenshot im Ticket ist der schnellste Weg, den Fall an einen Kollegen zu übergeben. Niemand in dieser Geschichte tut etwas Unvernünftiges, und am Ende des Nachmittags steht die Nummer in einem Chat-Log und in einem Ticket-Anhang.

Zwei Schreibtische weiter baut Jordan aus dem Data-Team einen Bericht auf `ledgerly_analytics` und bemerkt, dass die Tabelle `customers` dort Kontaktnamen, E-Mail-Adressen und Telefonnummern enthält, die kein Bericht braucht. Robin, in Rufbereitschaft, liest ein Anwendungs-Log, um herauszufinden, warum eine Erinnerung fehlgeschlagen ist, und findet die E-Mail-Adresse des Empfängers in der Fehlermeldung. Jede dieser Offenlegungen ist klein, und alle drei haben dieselbe Form: Daten sind an einem Ort angekommen, an dem niemand entschieden hat, dass sie sein sollen.

Dieses Kapitel fügt einen Schritt des Zielzustands hinzu: Sensible Spalten werden klassifiziert, standardmäßig maskiert oder vorenthalten, und die Analytics-Kopie enthält keine rohen Restricted-Daten. Die Frage, die es beantwortet, lautet: Was muss jede Person und jede Kopie tatsächlich sehen, und wie baut man das System so, dass es genau das zeigt und nicht mehr?

Im Modell aus fünf Teilen, das dieses Buch verwendet (**Identität, Grant, Pfad, Ausführung, Aufzeichnung**), arbeitet das Kapitel an zwei Teilen. Der Grant legt fest, welche Spalten eine Rolle überhaupt lesen darf, und die Ausführung bestimmt, in welcher Form ein Wert über den Pfad zurück-

kommt. Nichts davon hängt davon ab, wie ein Credential zu einer Person gelangt ist; ein perfekt abgesicherter Login, der eine vollständige Kontonummer liest, hat sie dennoch offengelegt.

Das Problem und warum es schwierig ist

Die Kontrollen aus den früheren Kapiteln entscheiden, wer welche Tabelle und welche Spalten erreicht. Sie wissen nicht, warum eine Spalte wichtig ist. Ein Grant auf `customers` behandelt ein Status-Flag und eine Kontakt-E-Mail gleich, und das Schema nennt beide `text`. Nichts in einer Spaltendefinition sagt, dass `invoice_lines.description` ein Freitext ist, den jemand mit einem Namen gefüllt haben kann, oder dass `iban` ein Wert ist, dessen Offenlegung sich nicht dadurch rückgängig machen lässt, dass man etwas rotiert.

Die zweite Schwierigkeit ist, dass Menschen einen Teil der Daten brauchen, und eine Regel, nach der niemand sie sehen darf, erzeugt Umgehungen. Der Support muss Fragen zu Auszahlungskonten beantworten. Die Rufbereitschaft muss herausfinden, warum eine Erinnerung fehlgeschlagen ist. Das Data-Team muss Kunden nach Segmenten zählen. Zeigt der genehmigte Pfad nichts, tauchen die nicht genehmigten auf: Screenshots, Ad-hoc-Exporte, ein Dauer-Grant „bis das Projekt endet“. Die nützliche Frage lautet nicht, ob eine Person einen Wert sehen darf, sondern welche Form des Werts die Aufgabe braucht: den ganzen Wert, seine letzten Ziffern, ein Ja oder Nein oder eine Zählung.

Die dritte Schwierigkeit ist, dass Daten nicht in der Tabelle bleiben. Sie werden in einen Analytics-Speicher kopiert, in Logzeilen geschrieben, in Fehlermeldungen wiedergegeben, in Tickets eingefügt und in Exporten verschickt. Jede Kontrolle auf dem Primary endet an der Grenze der Datenbank, und der Wert setzt seinen Weg fort. Deshalb muss die Antwort ins System eingebaut werden und kann nicht bei einer Richtlinie stehen bleiben. Eine Richtlinie, die lautet „gehen Sie sorgfältig mit personenbezogenen Daten um“, kann von keiner Migration-Pipeline, keinem geplanten Load und keinem Log-Formatter ausgeführt werden. Die Entscheidungen müssen in den Dingen stehen, die laufen: im Schema, in den Grants, im Load und im Logging-Code.

Datenschutz- und Zahlungskarten-Regelwerke wie GDPR, KVKK oder PCI DSS stellen tendenziell eine Reihe ähnlicher Fragen: Welche personenbezogenen Daten halten Sie, warum, wohin gehen sie, wer kann sie sehen und wie lange. Eine im Schema geführte Klassifizierung ist die Form, in der Sie sie beantworten können. Ihre Rechts- und Compliance-Verantwortlichen besitzen die Auslegung jedes Regelwerks; dieses Kapitel beschreibt nur die Technik, mit der sich die Antworten finden lassen. Der Katalog ist ein Nachweis dessen, was das System tut, und keine Behauptung, dass es ein Regelwerk erfüllt; Kapitel 14 kommt darauf zurück, welche Nachweise ein Zugriffs-Review sammeln kann.

Klassifizierung, die im Schema lebt

Die **Datenklassifizierung** ordnet Daten benannten Klassen zu, und jede Klasse bestimmt den Umgang. Ledgerly verwendet vier.

Klasse	Was sie bedeutet	Beispiele bei Ledgerly	Standardumgang
Restricted	Eine Offenlegung würde Kunden oder das Unternehmen auf schwer umkehrbare Weise schädigen	`iban`, `account_holder_name`, `token_value`	Für `human_readonly` vorenthalten; nie in einem Just-in-Time-Antrag (JIT); nur über einen Anwendungspfad änderbar; in der Analytics-Kopie nicht vorhanden
Confidential	Personenbezogene oder geschäftliche Daten über Kunden und Kontakte	`contact_email`, `billing_address`, `provider_reference`, `invoice_lines.description`	Für `human_readonly` maskiert oder vorenthalten; unmaskiert nur in einem `SELECT` per JIT mit Genehmiger; in der Analytics-Kopie maskiert oder entfernt, außer den Anzeigehinweisen, die zum Anzeigen da sind (`provider`, `expires_on`; siehe die Entscheidungstabelle weiter unten)
Internal	Betriebsdaten ohne personenbezogenen Inhalt	`amount_minor`, `status`, `due_on`, `account_code`	Lesbar für die Rollen, die sie brauchen; in die Analytics kopiert

Klasse	Was sie bedeutet	Beispiele bei Ledgerly	Standardumgang
Public	Unbedenklich zu veröffentlichen	Keine unter den acht Tabellen	Nur Integrität

Die Zeile Public ist absichtlich leer. Es gibt sie, damit die Frage „Darf das veröffentlicht werden?“ eine andere Antwort hat als Schweigen.

Vier Regeln machen die Klassen brauchbar. Das ist der Teil, den ich aufschreiben würde, bevor ich irgendein Tool auswähle.

- **Klassifizieren Sie Spalten, nicht Tabellen.** `customers` enthält ein Status-Flag und eine Telefonnummer, und beide brauchen einen unterschiedlichen Umgang. Nur für Inventarzwecke erhält eine Tabelle die höchste Klasse unter ihren Spalten.
- **Klassifizieren Sie Freitext für den schlimmsten Fall.** Eine Spalte, in die Menschen tippen, enthält schließlich alles, was Menschen tippen. `invoice_lines.description` ist Confidential, weil sie personenbezogene Daten enthalten kann, und kein Filter kann erkennen, welche Zeilen das tun.
- **Ein abgeleiteter Wert erbt die Klasse seiner Quelle, bis jemand das Gegenteil begründet.** Ein Hash mit Schlüssel über eine `iban` ist Restricted, bis eine Person begründet hat, dass er es nicht ist. `payment_tokens.last_four` ist Confidential und nicht Restricted, weil die Spalte zum Anzeigen da ist, und diese Entscheidung wurde einmal getroffen und aufgeschrieben.
- **Unbekannt ist Confidential.** Eine Spalte, die niemand klassifiziert hat, wird als Confidential behandelt und als vorläufig markiert, damit die unfertige Arbeit sichtbar ist und der Standard der geschlossene ist.

Ein Katalog im selben Repository wie die Migrationen

Eine Klassifizierung, die in einem Wiki lebt, driftet, weil nichts, was das Schema ändert, das Wiki liest. Ich würde sie als Datei in dem Repository führen, das die Migrationen enthält, und sie im selben Pull Request än-

dern, der die Spalte hinzufügt. Manche Engines erlauben auch Labels oder Kommentare an Spalten; beide Formen funktionieren, solange die Pipeline sie lesen kann.

```
// 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
```

Der Katalog ermöglicht zwei Prüfungen. Die erste läuft gegen jede Migration, neben den Flags aus Kapitel 11. Anders als diese Flags, die einen Reviewer bitten hinzuschauen, blockiert diese den Merge, denn eine Spalte zu klassifizieren kostet eine Zeile, und eine fehlende Zeile ist genau das, was die ganze Kontrolle abfangen soll.

```
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)
```

Hier bedeutet `inRestrictedHome` eine Tabelle, in der Grants die Spalte vorenthalten können: eine Haupttabelle mit Grants auf Spaltenebene oder, auf Engines ohne diese, die separaten Secrets-Tabellen aus Kapitel 3. Ist

die Person, die die Spalte hinzufügt, unsicher, besteht der Ausweg nicht darin, die Prüfung zu überspringen, sondern die Spalte als Confidential mit leerem `reviewed_on` zu klassifizieren und das nächste Zugriffs-Review entscheiden zu lassen.

Die zweite Prüfung läuft nach Zeitplan und vergleicht das Live-Schema mit dem Katalog, so wie Kapitel 11 das Live-Schema mit den Migrationen vergleicht. Eine Spalte, die in einer schlechten Nacht von Hand angelegt wurde, erscheint als Befund mit einer verantwortlichen Person, und der Befund wird durch eine Migration und eine Katalogzeile gelöst, nicht durch eine stille Änderung.

Minimierung: Die Daten, die Sie nicht halten

Minimierung heißt, nicht zu erheben, zu kopieren oder aufzubewahren, was niemand braucht. Sie ist die günstigste Kontrolle in diesem Kapitel, denn ein Wert, der nie gespeichert wurde, hat keine Maske, die gepflegt werden muss, keine Kopie, die man finden muss, und keine Zeile, die man schwärzen muss. Jede andere Kontrolle hier setzt voraus, dass die Daten existieren; diese fragt, ob sie existieren sollten.

Für jede sensible Spalte würde ich vier Fragen in dieser Reihenfolge stellen.

1. Braucht irgendeine Funktion oder Pflicht diesen Wert? Lautet die Antwort „er könnte nützlich sein“, lautet die Antwort Nein.
2. Könnten wir stattdessen einen Ersatzwert halten? Ledgerly hält `payment_tokens`, vom Provider ausgegebene Tokens, die für Zahlungsinstrumente stehen, und nicht die Instrumente selbst. Diese eine Entscheidung entfernt eine ganze Klasse von Werten aus dem Schema.
3. Könnten wir weniger davon halten? Ein Anzeigehinweis wie `last_four` lässt Support und Kunden ein Instrument erkennen, ohne dass das Instrument vorliegt.
4. Könnten wir es kürzer halten? Die Kontaktdaten eines geschlossenen Kunden haben einen Zweck, bis die letzte Rechnung beglichen ist und die Widerspruchsfristen verstrichen sind; danach werden sie nur noch für einen kurzen, festgelegten Zeitraum aufbewahrt und dann gelöscht.

Zwei Entscheidungen darüber, wo Daten liegen, wiegen schwerer als jede Aufbewahrungsregel. Die erste lautet: **Append-only-Tabellen sollten keine personenbezogenen Daten enthalten.** `ledger_entries` enthält `amount_minor` und `account_code`, beide Internal, und deshalb kann eine Korrektur ein neuer Eintrag sein: Nichts in der Tabelle muss je gelöscht werden, um einem Ersuchen zu einer Person nachzukommen. `audit_events` verweist aus demselben Grund über eine opake `subject_id` auf sein Subjekt. Die zweite ist Freitext. Ein strukturiertes Feld mit festem Wertevorrat lässt sich klassifizieren und maskieren; ein Notizfeld lässt sich beides nicht. Wo eine Funktion also statt eines Satzes einen Code, einen Status oder eine Produktreferenz verwenden kann, würde ich diese wählen.

Aufbewahrung ist ein Job, kein Satz in einer Richtlinie. Etwas muss nach Zeitplan laufen, Zeilen finden, deren Frist abgelaufen ist, und sie löschen oder anonymisieren, und dieses Etwas braucht eigene enge Privilegien, eigene Audit-Einträge und eine verantwortliche Person. Ich weiß nicht, welche Frist für Ihre Daten richtig ist, und dieses Buch weiß es auch nicht. Sie hängt davon ab, was Ihre Verträge, Ihre Aufsichtsbehörden und Ihr eigener Widerspruchsprozess verlangen, und Ihre Rechtsabteilung muss das beantworten, bevor ein Entwickler den Job schreibt. Was ein Entwickler sagen kann: Auch Backups, Exporte und die Analytics-Kopie enthalten alte Daten, und eine Löschung, die nur den Primary erreicht, ist nicht abgeschlossen. Kapitel 13 kommt darauf zurück.

Maskierung: dynamisch und statisch

Maskierung ersetzt oder verschleiert einen Wert, sodass der echte nicht sichtbar ist, während die Form brauchbar bleibt. **Redaction** ist die grobe Variante: Der Wert wird aus einer Ausgabe oder einem Log entfernt. Die beiden Arten der Maskierung unterscheiden sich darin, wo sie wirken.

	Dynamische Maskierung	Statische Maskierung
Was sich ändert	Was eine Abfrage zurückgibt	Die Kopie selbst
Wann	Beim Lesen, pro Leser	Beim Laden, einmal
Rohwert weiterhin gespeichert	Ja, im selben Speicher	Nicht in der Kopie

	Dynamische Maskierung	Statische Maskierung
Typische Verwendung	`human_readonly` auf einem Replikat	Die Analytics-Kopie, Staging-Daten
Wie sie versagt	Engine-Verhalten, Filter, direkte Grants	Eine veraltete oder zu stark gekürzte Kopie

Dynamische Maskierung für Menschen

`human_readonly` ist der Standard, mit dem jede Session beginnt, daher ist seine Maskierung die, der die meisten Menschen begegnen. Die Notation unten ist illustrativ, weil Engines Maskierung sehr unterschiedlich ausdrücken: Manche bieten eine Maske auf der Spalte, manche ein Privileg zum Aufheben der Maske, manche eine feste View pro Tabelle.

```
// 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;
```

Die Wahl der Funktion folgt daraus, wofür die Spalte da ist. Eine Technik ist an sich weder besser noch schlechter; jede hat ihren Platz und ihre Art zu versagen.

Technik	Was ein Leser erhält	Geeignet für	Wo sie nicht mehr hilft
Teilmaskierung	Die ersten oder letzten Zeichen	Ein Bildschirm, ein eingefügtes Ergebnis	Kurze Werte zeigen das meiste von sich selbst
Redaction	Eine feste Markierung oder nichts	Adressen, Freitext	Kann nichts bestätigen
Pseudonym mit Schlüssel	Ein stabiler Ersatzwert	Joins über Loads hinweg	Die Rotation des Schlüssels bricht die Joins
Vergröberung	Ein gröberer Wert, etwa ein Land oder ein Monat	Analytics	Kombinierte Spalten können eine Person herausgreifen (Kapitel 13)

Ein einfacher Hash steht nicht auf der Liste. Ein Hash über einen Wert aus einer Menge, die so klein ist, dass sich jeder Kandidat ausprobieren lässt, etwa eine Telefonnummer, lässt sich durch Ausprobieren umkehren; ein Pseudonym braucht also einen Schlüssel, der außerhalb der Datenbank liegt.

Nicht jede Confidential-Spalte wird maskiert. Eine Spalte, die zum Anzeigen da ist, bleibt lesbar: `payment_tokens.provider`, `last_four` und `expires_on` sind die Anzeigehinweise, an denen eine Person ein Instrument erkennt, und sie zu maskieren würde die Leute nur zu dem Restricted-Wert drängen, den sie ersetzen. Kontaktfelder, die Rechnungsadresse und `provider_reference` identifizieren eine Person oder eine Transaktion und werden maskiert. `invoice_lines.description` wird vorenthalten.

Daraus ergibt sich die Regel, von der Kapitel 8 abhängt. `human_readonly` liest bereits, daher wird ein JIT-Tripel mit `SELECT` nur für eine Confidential-Spalte gebraucht, die die Rolle maskiert oder vorenthält, und was es hinzufügt, ist diese Spalte unmaskiert. Restricted-Spalten erscheinen nie in einem JIT-Tripel, daher ändert sich für sie hier nichts. Der Antrag und der Bildschirm des Genehmigers sollten deshalb eine ausdrückliche Zeile tragen, „Unmasked columns: `contact_email`“, getrennt von der Tabellenliste, damit der Genehmiger sieht, welche Masken der Antrag aufhebt, und nicht nur, welche Tabellen er berührt.

> **Warnung:** Prüfen Sie, wie Ihre Engine einen Login behandelt, der eine Spalte über `human_readonly` liest und zusätzlich einen direkten Spalten-Grant hält. Setzt sich die Maske durch, zeigt ein `SELECT` per JIT nichts Neues, und die Genehmigung bedeutet nichts. Setzen sich für jeden Leser die Rohwerte durch, ist die Maske Dekoration. Testen Sie beide Fälle an einer Kopie, bevor Sie sich auf einen verlassen.

Zwei Grenzen gehören neben die Technik. Erstens gilt eine Maske auf vielen Engines für das, was eine Abfrage zurückgibt, nicht für das, was ihre Bedingungen vergleichen; ein Leser, der auf einer maskierten Spalte filtern darf, kann also über Zählungen und Treffer etwas über ihre Werte lernen. Betrachten Sie dynamische Maskierung als Schutz vor beiläufiger Offenlegung, etwa durch einen Bildschirm, einen Screenshot oder ein eingefügtes Ergebnis. Zweitens tut die Maskierung einer Ausgabe nichts für den gespeicherten Wert, der in jedem Backup, jedem Replikat und jedem Dump des Primary erhalten bleibt.

Aus der ersten Grenze folgt eine Entscheidungsregel für Rückschlüsse. Fragen Sie, wie günstig ein Leser den Wert durch Filtern wiederherstellen könnte: Eine Handvoll Gleichheitsproben auf `contact_email` ist günstig, und eine Zählung von Kunden nach einem maskierten Telefonpräfix ist es ebenfalls. Wo es günstig ist und der Leser den Wert nicht erfahren darf, enthalten Sie die Spalte vor, statt sie zu maskieren, oder verbieten Sie für diese Rolle Prädikate auf der maskierten Spalte. Wo es teuer ist, etwa beim Ausprobieren eines langen freien Werts Zeichen für Zeichen über Tausende protokollierter Anweisungen, ist Maskierung plus Aufzeichnung der Anweisungen eine vernünftige Kontrolle.

Maskierung im Ergebnis-Stream

Manche Zugangspfade können sich nicht darauf verlassen, dass die Engine maskiert. Ein Gateway zwischen der Person und der Datenbank kann Regeln auf den Ergebnis-Stream selbst anwenden, bevor Zeilen den Bildschirm oder eine Ergebnisdatei erreichen. Das hilft dort, wo die Engine keine Maskierungsfunktion hat, und die Regeln können für jede dahinterliegende Engine dieselben sein.

Eine Stream-Regel setzt voraus zu wissen, aus welcher Spalte ein Wert stammt, und ein Alias wie `SELECT api_key AS k` unterläuft Regeln, die auf den Spaltennamen passen. Tools, die Ergebnisse maskieren, darunter QueryProxy, fügen Regeln hinzu, die zusätzlich auf den Wert passen; das ist eine zweite Schicht, nie ein Ersatz für den Grant. Zum Zeitpunkt der Niederschrift dokumentiert es diese Einschränkung namensbasierter Regeln selbst, und andere Tools gehen ähnlich vor. Die Folge für Ledgerly ist, dass eine Stream-Maske hinter den Spalten-Grants aus Kapitel 3 sitzt und eine Spalte, die eine Rolle nie sehen darf, durch den Grant vorenthalten wird und nicht einem Muster überlassen bleibt.

> **Hinweis:** Auf Engines ohne Grants auf Spaltenebene verwendet Ledgerly den Fallback aus Kapitel 3: Die Restricted-Spalten liegen in `bank_account_secrets` und `payment_token_values`. Die Maskierung von `human_readonly` bedeutet dann meist eine feste maskierte View pro Tabelle, die einmal per Migration erstellt wird und `ledgerly_owner` gehört, wobei die Rolle `SELECT` auf die View hält und nicht auf die Basistabelle. Ein Grant auf Tabellenebene für die Basistabelle in einem JIT-Antrag zeigt jede Nicht-Restricted-Spalte roh, auch die Confidential-Spalten, und der Genehmiger hat genau das akzeptiert. Das ist eine feste View, keine pro Antrag erstellte View, und sie ist ein Maskierungsmechanismus für Confidential-Spalten. Sie ist kein zweiter Fallback für Restricted-Spalten: Der einzige Fallback für diese sind die Secrets-Tabellen.

Statische Maskierung für die Analytics-Kopie

Bei der Analytics-Kopie ist Maskierung am günstigsten und am nützlichsten, weil der Rohwert ganz aus der Kopie herausbleiben kann. Jordans Bericht braucht Kundensegmente, Zählungen und Beträge. Er braucht keine Namen, E-Mail-Adressen oder Telefonnummern, daher ist für die meisten Confidential-Spalten die richtige Behandlung `drop`, und eine Transformation ist für die wenigen da, die Analysten nachweislich verwenden.

Der Load ist ein geplanter Job, kein Service, und er läuft unter einem eigenen Login und der Rolle `app_analytics_load`, wie jeder andere Principal im Zugriffs-Review. Die Rolle hält `SELECT` nur auf die Spalten, die der Katalog nicht als Restricted markiert; der Grant ist also die erste Absicherung

und der Katalog die zweite. Sie sind nicht unabhängig, denn der Grant wird aus den Restricted-Markierungen des Katalogs abgeleitet: Eine fälschlich als Internal klassifizierte Spalte passiert beide. Was der Grant hinzufügt, ist, dass eine als Restricted markierte Spalte selbst dann nicht gelesen werden kann, wenn ein Transformationseintrag falsch ist; die unabhängige Prüfung ist die Schema-Assertion in `verifyMasked`, die die Kopie mit dem Plan vergleicht. Innerhalb dieses Grants entscheidet der Load Spalte für Spalte anhand des Katalogs, und er schlägt geschlossen fehl.

```
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` nimmt zwei Prüfungen vor. Es prüft per Assertion das Schema: Die Spaltenmenge des Ziels muss dem Plan entsprechen, und es darf keine Restricted-Spalte und keine entfernte Spalte darin existieren. Dann zieht es Stichproben von Zeilen und schlägt fehl, wenn eine transformierte Spalte noch ihrem Quellwert entspricht. Die Schema-Assertion fängt eine Spalte ab, die über einen Weg ankam, den der Plan nicht aufführte; die Stichprobe fängt eine Transformation ab, die falsch konfiguriert wurde. Keine von beiden ist ein Beweis, dass die Kopie nichts Sensibles enthält, denn die Stichprobe liest nur einen Teil der Zeilen.

Der Preis des geschlossenen Fehlschlagens ist sichtbar: Wenn ein Entwickler eine Spalte hinzufügt und den Katalog vergisst, hätte die Migrationsprüfung das stoppen sollen, und tat sie es nicht, wird die Analytics-Tabelle ohne diese Spalte geladen, und das Data-Team bemerkt eine Lücke. Ich würde das akzeptieren. Ein Bericht mit fehlender Spalte ist ein Ticket; eine Kopie mit einer unmaskierten Spalte ist eine Offenlegung, die niemand bemerkt.

Ein Pseudonym mit Schlüssel für Joins verdient eine eigene Entscheidung. Es lässt Analysten einen Kunden über Loads hinweg verfolgen, ohne zu wissen, wer der Kunde ist, und es funktioniert nur, solange der Schlüssel außerhalb der Reichweite des Data-Teams und unverändert bleibt. Eine Rotation des Schlüssels ändert jedes Pseudonym, sodass die Historie nicht mehr joint. Für viele Teams erledigt die opake `customer_id` diese Aufgabe bereits, und ein zweiter Ersatzwert fügt einen zu schützenden Schlüssel ohne Gewinn hinzu.

Tokenisierung und Verschlüsselung auf Anwendungsebene

Tokenisierung ersetzt einen Wert durch einen Ersatzwert, der nur von einem kontrollierten Service zurückgeführt werden kann. Sie ist keine Maskierung, weil die Zuordnung existiert, und meist auch keine Verschlüsselung, weil der Ersatzwert normalerweise gar nicht aus dem Wert berechnet wird. Verfahren, die ihn unter einem Schlüssel aus dem Wert berechnen, sind Verschlüsselung unter anderem Namen und brauchen dieselbe Sorgfalt beim Schlüssel. Ledgerly profitiert davon bereits an einer Stelle: `token_value` ist das Token eines Providers, der Provider hält die Zuordnung, und `payments-service` ist der einzige Aufrufer. Die Ledgerly-Datenbank hält das Instrument deshalb nie.

Einen zweiten Tokenisierungs-Service für Bankkontonummern zu bauen, ist ein größerer Schritt. Er fügt dem Zahlungspfad ein System hinzu, das verfügbar sein muss, wenn Zahlungen es sind, er braucht eine Liste der Aufrufer, die ein Token zurückführen dürfen, und jede Zuordnung ist ein Ereignis, das jemand prüfen können sollte. Bei Ledgerlys Größe würde ich ihn nicht bauen. Die Spalten einzuschränken, den einen Service einzuschränken, der sie liest, und die Werte zu verschlüsseln würde den größ-

ten Teil des Schutzes mit weniger Betriebsaufwand bieten. Verlangt ein regulierter Partner oder ein Zahlungsdienstleister ihn, entscheidet diese Anforderung.

Verschlüsselung auf Anwendungsebene heißt, dass der Service einen Wert verschlüsselt, bevor die Datenbank ihn sieht, sodass Rollen, Replikate, Backups und Dumps alle Chiffretext enthalten. Das ist ihre besondere Stärke: Sie schützt den Wert in den Datenkopien, die ein Grant nicht erreicht (Kapitel 13). Verschlüsselung durch die Speicherschicht, auf der Platte oder in einem Snapshot, schützt vor einem verlorenen Volume und tut nichts gegen eine Abfrage, und Teams verwechseln sie oft mit Zugriffskontrolle. Verschlüsselung auf Anwendungsebene ist näher an einer echten Grenze, und sie ist eine schwerere Verpflichtung.

- **Der Schlüssel entscheidet, wer lesen kann.** Die Daten sind nur so gut geschützt wie ihr Schlüssel. Geben Sie die Entschlüsselungsberechtigung der Workload Identity von `payments-service` und sonst niemandem, sodass die Entscheidung eine Frage ist, wer entschlüsseln kann und wodurch. Schlüsselhierarchien, Rotation und Schlüssel-Backup sind eine eigene Disziplin, und die Dokumentation des Key-Service behandelt sie besser, als dieses Buch es kann.
- **Die Spalte verliert ihre Abfragefunktionen.** Auf Chiffretext können Sie nicht wie gewohnt suchen, sortieren, nach Bereichen filtern oder Eindeutigkeit erzwingen; wählen Sie also Spalten, die immer nur über einen Bezeichner zurückgelesen werden.
- **Ein Service, der auf Anforderung entschlüsseln kann, kann dazu aufgefordert werden.** Die Verschlüsselung verlagert die Grenze auf den Service, daher werden seine Aufrufer, seine Rate und seine Audit-Einträge zur Kontrolle, und der Audit-Eintrag wird vor der Entschlüsselung geschrieben, wie unten.

```
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)
```

Die Matrix ändert sich nicht, wenn Sie verschlüsseln. `human_breakglass` liest die Restricted-Spalten weiterhin, aber was die Rolle liest, ist Chiffretext, und zum Entschlüsseln braucht es eine Schlüsselberechtigung, die die Rolle nicht trägt. Ich würde im Voraus entscheiden, ob ein Incident, der Klartext braucht, um Schlüsselzugriff bitten darf, mit welchem Genehmiger, und das in das Break-Glass-Runbook aus Kapitel 8 schreiben. Diese Entscheidung dem Incident zu überlassen heißt, eine Regel zu improvisieren.

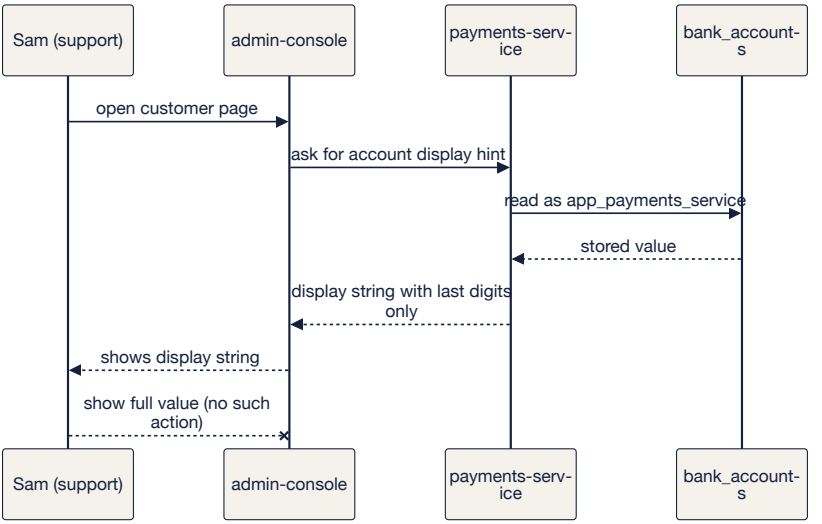
Ohne Einblick arbeiten: der Support-Fall

Zurück zu Sam. Das Ziel ist, dass Sam dem Kunden vollständig antworten kann und den vollen Wert nie sieht. Die meisten Support-Fragen brauchen am Ende weniger als den Wert.

Was der Support tun muss	Was er dafür sehen muss	Wie es bereitgestellt wird
Dem Kunden sagen, welches Konto hinterlegt ist	Die letzten Ziffern	`payments-service` gibt einen Anzeigetext zurück; oder der Hinweis wird beim Schreiben des Kontos gespeichert, wie `last_four` bei Tokens
Prüfen, dass die schreibende Person der Kontakt ist	Ein Ja oder Nein	Eine enge Prüfung im Service, der den Grant bereits hält, die vergleicht, was der Kunde gesagt hat, mit dem gespeicherten Wert und „Treffer“ oder „kein Treffer“ antwortet
Die Rechnungen des Kunden ansehen	Status, Beträge, Fälligkeitsdaten, nur für diesen Kunden	`app_admin_console` liest Internal-Spalten von `invoices`, und die Zeilenregel zum Kunden des Falls aus Kapitel 3 schränkt die Zeilen ein

Was der Support tun muss	Was er dafür sehen muss	Wie es bereitgestellt wird
Eine falsche Kontonummer korrigieren	Nichts, und nicht von Hand	Der Anwendungspfad: `admin-console` fragt `payments-service`, eine zweite Person genehmigt, und die Änderung wird in `audit_events` aufgezeichnet

Das Diagramm zeigt die erste Zeile. Achten Sie auf den letzten Pfeil: Eine Aktion, die nicht existiert, ist eine stärkere Kontrolle als eine, die existiert und abgelehnt wird.



Ich würde zu Beginn keinen „Reveal"-Button bauen. Ein Reveal, das einen Grund verlangt und den Klick protokolliert, klingt vernünftig, und innerhalb eines Monats ist das Klicken eine Gewohnheit und sein Log etwas, das niemand liest. Entsteht ein echter Bedarf, machen Sie daraus einen Antrag mit Genehmiger, wie Kapitel 8 es für jedes Lesen unmaskierter Confidential-Daten tut, und zählen Sie, wie oft er genutzt wird. Eine Funktion, die täglich genutzt wird, sagt Ihnen, dass der Anzeigeform etwas fehlt.

Wer Rufbereitschaft hat, folgt derselben Regel unter anderem Druck. Robin, der oder die einen fehlgeschlagenen Erinnerungsversand debuggt, muss wissen, dass eine Empfängeradresse abgelehnt wurde, und braucht die Adresse selten. Braucht Robin doch die unmaskierte `contact_email`, ist das ein `SELECT` per JIT, das die Spalte benennt, mit einem Genehmiger,

der die Spaltenliste liest. Lesezugriffe in der Rufbereitschaft werden nie automatisch genehmigt: Eine Regel, die Confidential-Spalten überspringt, genehmigt einen Antrag, der nichts gewährt, und eine Regel, die sie einschließt, entfernt die eine Prüfung, die Bedeutung trägt. Der Preis ist eine Wartezeit zu einer schlechten Stunde, und man zahlt ihn ab, indem man den Bedarf an dem Wert in den häufigen Fällen beseitigt, was der nächste Abschnitt behandelt.

Sensible Daten in Logs und Fehlermeldungen

Logs sind die Datenkopie, die niemand entwirft. Sie werden von vielen Komponenten geschrieben, von vielen Menschen gelesen, lange aufbewahrt und mit Tools durchsucht, die nie Teil des Zugriffsmodells waren. Vier Stellen verursachen den größten Teil der Lecks.

- **Anwendungs-Logs und Fehlermeldungen.** Eine aus einem Wert gebaute Nachricht wie „could not send to the address“ bringt den Wert ins Log, in den Error-Tracker und manchmal auf den Bildschirm eines Kunden. Treiber und ORMs können außerdem gebundene Parameter ins Log schreiben, wenn das Debug-Logging eingeschaltet ist.
- **Datenbank-Logs.** Das Statement-Logging kann Literale aufzeichnen, und ein Slow-Query-Log bewahrt den Text einer Abfrage samt den darin enthaltenen Werten auf.
- **Aufzeichnungen der Anweisungen.** Das Access-Gateway zeichnet jede Anweisung auf, die eine Person sendet (Kapitel 7), und wer einen Wert in eine Bedingung tippt, hat diesen Wert in die Aufzeichnung gebracht.
- **Tickets und Chat.** Ein eingefügtes Ergebnis, um ein Problem zu erklären, ist eine Kopie mit dem falschen Publikum und ohne Enddatum.

> **Warnung:** Ein Anweisungs-Log enthält Literale, daher sind das Audit-Log aus Kapitel 14 und die Anweisungsaufzeichnungen des Gateways selbst Speicher sensibler Daten. Zeichnen Sie Anweisungen als Fingerprints mit geschwärzten Literalen auf, klassifizieren Sie den Speicher mindestens als Confidential, beschränken Sie, wer ihn liest, und geben Sie ihm eine Aufbewahrungsfrist, sonst wird die Kontrolle, die die Daten überwacht, zu einer weiteren Kopie von ihnen. Den Mechanismus behandelt Kapitel 14.

Meine bevorzugte Regel für das Anwendungs-Logging ist eine Allowlist, kein Scrubber. Ein Scrubber sucht nach Mustern, die sensiblen Werten ähneln, und entfernt sie; er fängt die Formen ab, an die sein Autor gedacht hat, und übersieht den Rest. Eine Allowlist benennt die Felder, die erscheinen dürfen, und verwirft alles andere, sodass ein neues Feld stumm bleibt, bis jemand entscheidet, es ins Log zu schreiben.

```
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
```

Fehlermeldungen folgen derselben Disziplin. Ein Fehler trägt einen Code und Bezeichner, etwa eine Rechnungs-ID, eine Kunden-ID und einen Zustellstatus, und nicht den Wert, der fehlgeschlagen ist. So braucht auch Robins Erinnerungsfall die Rohadresse nicht mehr: Enthält der Fehlerdatensatz einen Status wie „recipient rejected“ und die Rechnungs-ID, beantwortet das Log die Frage ohne die Adresse, und die Aufzeichnung des Gateways darüber, wer die Adresse gelesen hat, falls jemand sie gelesen hat, bleibt an einer Stelle.

Redaction in der Log-Pipeline ist ein Auffangnetz, und sie wird Dinge übersehen. Die Prüfung, die ich nicht überspringen würde, ist ein Canary in Tests: die Anwendung mit einem Platzhalter erkennbarer Form in jedem

sensiblen Feld laufen lassen und den Build fehlschlagen lassen, wenn er in irgendeiner Log-Ausgabe auftaucht. Er testet die Allowlist dort, wo es zählt, und kostet ein paar Zeilen. Das Debugging wird langsamer, wenn das Log Bezeichner statt Werte enthält, und der `dropped_field_count` ist der Ort, an den die daraus folgenden „Warum kann ich das nicht sehen"-Meldungen gehen, statt dass ein Entwickler das Parameter-Logging in der Produktion wieder einschaltet.

Ledgerlys Spaltenentscheidungen

Die Tabelle unten fasst das Kapitel zusammen. Sie ist das Entscheidungsprotokoll für die sensiblen Spalten der acht Tabellen, in einer Form, die ein Zugriffs-Review lesen kann. In dieser Spalte bedeuten „Vorenthalten" und „Kein Zugriff" beide, dass die Rolle keinen Grant auf die Spalte hält, sodass eine Abfrage, die sie nennt, abgelehnt wird. Die Spalte `human_readonly` nennt den Standard einer Session; die JIT-Spalte nennt, ob ein Antrag die Spalte benennen darf.

Spalte	Klasse	`human_readonly`	`SELECT` per JIT	Analytics-Kopie
`bank_accounts.iban`, `account_holder_name`	Restricted	Kein Zugriff	Nie	Entfernt
`payment_tokens.token_value`	Restricted	Vorenthalten	Nie	Entfernt
`payment_tokens.provider`, `last_four`, `expires_on`	Confidential	Lesbar	Nicht nötig	`provider` und `expires_on` behalten, `last_four` entfernt
`customers.contact_name`, `contact_email`, `contact_phone`	Confidential	Maskiert	Benennt die Spalte, unmaskiert	Entfernt
`customers.billing_address`	Confidential	Geschwärzt	Benennt die Spalte, unmaskiert	Entfernt

Spalte	Klasse	`human_readonly`	`SELECT` per JIT	Analytics-Kopie
`payments.provider_reference`	Confidential	Letzte vier Zeichen	Benennt die Spalte, unmaskiert	Entfernt
`invoice_lines.description`	Confidential	Vorenthalten	Benennt die Spalte, unmaskiert	Entfernt
`amount_minor`, `currency`, `status`, `due_on`, `account_code`	Internal	Lesbar, wo die Matrix es gewährt	Nicht nötig	Kopiert

Zwei Einträge verdienen einen Kommentar. Die Restricted-Spalten werden aus der Analytics-Kopie entfernt und keiner menschlichen Rolle außer `human_breakglass` je angeboten; der einzige Weg, der sie ändert, ist der Anwendungspfad mit Genehmigung durch eine zweite Person (wie die Ausführungsrichtlinie gated Operationen dorthin leitet, behandelt Kapitel 9). Derselbe Pfad nimmt die Eigentümerverknüpfung auf: `customer_id` in `bank_accounts` und `payment_tokens` bleibt innerhalb der JIT-Obergrenze, weil sie nicht Restricted ist, doch eine Zeile zu einem anderen Kunden zu verschieben ändert, wessen Konto es ist; eine Änderung der Eigentümerverknüpfung ist daher gated und wird auf den Produktpfad geleitet, nie als gewöhnliche Datenkorrektur vorgenommen. Und wo die Engine keine Grants auf Spaltenebene hat, gilt dieselbe Tabelle weiterhin, mit einer Änderung: Die Restricted-Spalten liegen in den Secrets-Tabellen, die JIT-Spalte lautet „Tabellenebene“, und der Genehmigungsbildschirm sagt das in Worten.

Abwägungen und Kosten

Klassifizierung ist Pflegeaufwand. Ein Katalog, der am Tag seiner Erstellung korrekt ist, beginnt sofort zu altern. Jemand muss jede Klassenentscheidung verantworten, die vorläufigen prüfen und die Frage beantworten, die eine unklassifizierte Spalte aufwirft. Prüft niemand den Katalog, wird er zu der Art von Aufzeichnung, die vollständig und falsch ist, was schlimmer ist als eine ehrliche Lücke.

Maskierung kostet Debugging-Zeit. Robin kann die rohe E-Mail nicht sehen, daher braucht ein Bug, der von ihren genauen Zeichen abhängt, einen JIT-Antrag und einen Genehmiger. Diesen Preis nehme ich für den Zugriff in Kauf, den dieses Kapitel schließt. Er ist nicht gratis, und ein Team, das ihn jede Nacht zahlt, wird anfangen, einen Weg drumherum zu suchen; messen Sie also, wie oft das unmaskierte Lesen beantragt wird, und beheben Sie die Ursache.

Dynamische Maskierung hängt von der Engine ab. Das Verhalten bei Filtern, direkten Grants und Views unterscheidet sich, und eine Maske, die Sie auf Ihrer Engine nicht getestet haben, ist eine Hoffnung. Die View-basierte Variante fügt Objekte hinzu, die Migrationen mit den Basistabellen im Gleichschritt halten müssen.

Geschlossenes Fehlschlagen erzeugt Lücken. Der Analytics-Load, der unklassifizierte Spalten verweigert, lässt das Data-Team ohne eine Spalte, bis jemand sie klassifiziert. Benennen Sie, wer diese Wartezeit verantwortet, und halten Sie das Klassifizierungs-Review so schnell, dass die Lücke Stunden dauert, nicht Wochen.

Verschlüsselung auf Anwendungsebene nimmt Funktionen weg und fügt eine Abhängigkeit hinzu. Der Key-Service liegt auf dem Pfad jedes Lesens der Spalte, und sein Ausfall ist ein Zahlungsausfall. Weil der Audit-Eintrag zuerst kommt, stoppt auch ein unerreichbares Audit-Log diese Lesezugriffe, was eine bewusste Kopplung und eine weitere Verfügbarkeitsabhängigkeit ist. Der Verlust der Abfragefunktionen drängt die Suche auf andere Spalten, und das Schlüssel-Backup ist eine weitere Sache, die getestet werden muss.

Manche Kontrollen hier können zum Sicherheitstheater werden. Ein Button zum Aufheben der Maske, der immer geklickt wird, ein Katalog, den niemand prüft, ein Log-Scrubber, den niemand testet, und eine maskierte Spalte, die jeder Leser mit Filtern abtasten kann, erzeugen Reibung oder Papieraufwand und reduzieren nichts.

Wann Sie das alles nicht einsetzen. Ein kleines System mit einer Handvoll Tabellen und ohne personenbezogene Kundendaten braucht keine Katalogdatei, keinen maskierten Load und keinen Key-Service. Eine einseitige Liste der sensiblen Spalten, Grants auf Spaltenebene, die sie auslassen, und eine Log-Allowlist bringen den größten Teil des Nutzens. Fügen Sie

Maskierung hinzu, wenn ein zweites Publikum die Daten braucht, fügen Sie Verschlüsselung hinzu, wenn eine Kopie, um die sich nicht herumgranten lässt, zum Thema wird, und schreiben Sie auf, welches Ereignis Sie zum nächsten Schritt bewegen würde.

Häufige Fehler

- **Verschlüsselung im Ruhezustand als Zugriffskontrolle behandeln.** Sie verführt, weil sie eine einzige Einstellung ist und eine Checkliste erfüllt. Sie schützt vor einer verlorenen Platte und tut nichts gegen eine Abfrage. Behalten Sie die Grants, und verwenden Sie Verschlüsselung auf Anwendungsebene dort, wo eine Kopie Schutz braucht.
- **Im Client maskieren und nicht in der Datenbank.** Eine Maske in der Weboberfläche ist leicht gebaut, und eine direkte Verbindung, ein Export oder eine API-Antwort umgeht sie. Maskieren Sie dort, wo die Daten den Speicher verlassen.
- **Tabellen klassifizieren und nicht Spalten.** Ein Label pro Tabelle schützt das Status-Flag zu stark und die Telefonnummer zu schwach. Klassifizieren Sie jede Spalte und leiten Sie das Label der Tabelle daraus ab.
- **Neue Spalten standardmäßig auf Internal setzen.** Internal ist die Klasse mit der geringsten Reibung und wird deshalb zur Klasse von allem Ungeprüften. Machen Sie Confidential zum Standard und heben Sie es per Review auf.
- **Freitext mit Mustern maskieren.** Ein Muster findet die Zahlen und E-Mail-Adressen, die sein Autor erwartet hat, und übersieht einen Namen, der in eine Notiz getippt wurde. Enthalten Sie Freitext-Spalten breiten Lesergruppen vor.
- **Einen Hash ohne Schlüssel als Pseudonym verwenden.** Er sieht unumkehrbar aus. Bei einem Wert mit wenigen Möglichkeiten lässt er sich durch Ausprobieren umkehren. Verwenden Sie ein Pseudonym mit Schlüssel, der außerhalb der Datenbank bleibt, oder entfernen Sie die Spalte.
- **Die Produktion nach Staging kopieren „und hinterher maskieren“.** Das Fenster zwischen Kopie und Maske ist eine Kopie von allem. Maskieren Sie auf dem Weg, wie der Analytics-Load es tut, und behandeln Sie Staging als eine der Türen aus Kapitel 13.

- **Eine Restricted-Spalte in einen JIT-Antrag aufnehmen, um einen Schritt zu sparen.** Die Obergrenze verweigert sie aus gutem Grund, und die Versuchung ist nachts am stärksten. Restricted-Werte ändern sich nur über den Anwendungspfad mit Genehmigung nach dem Vier-Augen-Prinzip, und auf Engines ohne Grants auf Spaltenebene liegen sie in den Secrets-Tabellen.
- **Lesezugriffe in der Rufbereitschaft automatisch genehmigen.** Das beseitigt die Pause, in der eine Person prüft, welche Confidential-Spalte unmaskiert wird. Behalten Sie den Genehmiger, und verringern Sie, wie oft der Wert gebraucht wird.
- **Request-Bodys auf Error-Level ins Log schreiben.** Der Body ist das Nützlichste für einen Entwickler und enthält jedes Feld, das ein Kunde getippt hat. Schreiben Sie Bezeichner und Status ins Log, und lassen Sie eine Allowlist entscheiden, was sonst erscheint.

Zentrale Erkenntnisse

- Klassifizieren Sie jede Spalte, behandeln Sie Freitext für den schlimmsten Fall, und lassen Sie unklassifizierte Spalten standardmäßig Confidential sein, damit neue Daten geschlossen beginnen und die Arbeit, sie zu öffnen, sichtbar ist.
- Führen Sie den Klassifizierungskatalog in dem Repository, das die Migrationen enthält, und blockieren Sie einen Merge, der eine Spalte ohne Klasse hinzufügt, denn die Prüfung kostet eine Zeile und ihr Fehlen ist der Fehler.
- Minimierung kommt zuerst: Daten, die nie gespeichert wurden, brauchen keine Maske, und personenbezogene Daten, die aus Append-only-Tabellen herausgehalten werden, können nicht zum Löschproblem werden.
- Dynamische Maskierung passt zu `human_readonly` und schützt vor beiläufiger Offenlegung; statische Maskierung passt zur Analytics-Kopie, wo der Rohwert ganz entfernt werden kann und der Load bei allem Unklassifizierten geschlossen fehlschlägt.
- Geben Sie Menschen die Form eines Werts, die ihre Aufgabe braucht, etwa einen Anzeigetext oder ein Ja oder Nein, und halten Sie Restricted-Spalten aus jedem JIT-Antrag und jedem rohen SQL-Schreibzugriff heraus.

- Eine Maske im Ergebnis-Stream ist eine zweite Schicht hinter dem Grant, nie ein Ersatz dafür, und eine Load-Rolle, die Restricted-Spalten nicht lesen kann, lässt einen Katalogfehler geschlossen fehlschlagen.
- Logs, Fehlermeldungen und Anweisungsaufzeichnungen sind Datenkopien: Schreiben Sie Bezeichner über eine Allowlist ins Log, halten Sie Werte aus Nachrichten heraus, und klassifizieren Sie das Audit-Log selbst.

Die Kontrollen dieses Kapitels schützen Daten, solange sie in der Datenbank liegen und solange eine Abfrage sie zurückgibt. Das nächste Kapitel folgt den Daten nach draußen, in Backups, Dumps, Exporte und Staging-Kopien, wo keine dieser Rollen gilt.

Kapitel 13 — Backups, Exporte und andere Seitentüren

Stellen Sie sich einen Freitagnachmittag bei Ledgerly vor. Robin jagt einen Bug in den Rechnungserinnerungen, der nur mit realistischen Daten auftritt: Rechnungen mit ungewöhnlichen Fälligkeitsdaten, Kunden mit mehreren Kontakten, Positionsbeschreibungen mit Freitext, an den niemand gedacht hat. Staging enthält ein Dutzend von Hand angelegter Kunden, und auf keinem davon zeigt sich der Bug. Jemand erinnert sich, dass in einem gemeinsamen Speicher ein älterer Dump der Produktion liegt, und Alex, der die Datenbanken betreibt, spielt ihn in Staging ein, damit das ganze Team hinschauen kann.

Der Bug ist innerhalb einer Stunde gefunden. Der Dump bleibt im Ordner, und die wiederhergestellte Datenbank bleibt in Staging. Staging hat kein Access-Gateway, weil es nicht Produktion ist. Sein Login wird gemeinsam genutzt, jeder Entwickler kann ihn erreichen, CI-Jobs laufen dagegen, und ein Auftragnehmer, der im letzten Monat angefangen hat, hat dort einen Account. `invoice-worker` in Staging verwendet dieselben Einstellungen für ausgehende E-Mails wie in der Produktion, und sein Erinnerungs-Job ist eingeschaltet.

Niemand hat entschieden, Kundendaten offenzulegen. Jeder Schritt war eine vernünftige Antwort auf das Problem eines Nachmittags, und die Kontrollen aus den früheren Kapiteln schützen eine Datenbank, die nicht mehr die einzige Kopie ihres Inhalts hält.

Dieses Kapitel fügt den letzten Schritt des Zielzustands vor Teil V hinzu: Backups, Exporte und Staging-Kopien werden als Türen mit Verantwortlichen und Ablaufdatum behandelt. Die Frage lautet: Wenn Daten den Primary absichtlich verlassen, zur Wiederherstellung, für einen Dienstleister, für einen Test oder für eine Person, die hineinsehen muss, wie halten Sie die Kontrollen dann bei den Daten, statt bei der Datenbank, aus der sie stammen?

Im Modell aus fünf Teilen (**Identität, Grant, Pfad, Ausführung, Aufzeichnung**) ist eine Kopie ein Ort, an dem Grant und Pfad nicht mehr gelten, weil die Daten über eine Route hinausgelangt sind, die keines von beiden trägt. Die Aufgabe des Kapitels ist, dafür zu sorgen, dass die Kopie selbst einen Grant trägt: eine Leserliste, einen Verantwortlichen und ein Enddatum, dazu eine Aufzeichnung darüber, wer sie erstellt und wer sie entfernt hat.

Das Problem und warum es schwierig ist

Kapitel 6 hat eine **Seitentür** definiert als eine Datenkopie oder einen Exportpfad, der die Zugriffskontrollen des Primary umgeht. Replikate und die Analytics-Kopie sind Türen, die Sie bewusst bauen und benennen können. In diesem Kapitel geht es um den Rest: Backups und Snapshots, Dumps, CSV-Dateien, Staging-Datenbanken, ein paar tausend Zeilen auf einem Laptop, der Cache eines Dashboards, eine Datei im Posteingang eines Dienstleisters. Die meisten davon sind aus gutem Grund entstanden. Nur wenige haben einen Verantwortlichen.

Die Kernschwierigkeit besteht darin, dass Zugriffskontrollen in einem System leben und nicht in Daten. Der Primary hat ein Access-Gateway, einen Login pro Session, Grants auf Spaltenebene, Timeouts und ein Audit-Log. Ein Dump des Primary hat nichts davon, denn ein Dump ist eine Datei. Wer die Datei lesen kann, hat alles darin, auch Spalten, die das Rollenschema jeder menschlichen Rolle außer `human_breakglass` vorenthält. Ein Backup, das die Restricted-Spalten enthält, ist für die Leute, die es lesen können, ein Dauerzugriff auf diese Spalten, den kein Gateway ausgestellt hat und den kein Review auflistet.

Die naheliegende Antwort ist eine Richtlinie: Produktionsdaten werden nicht kopiert. Sie scheitert aus einem schlichten Grund. Kopieren ist der Weg, wie die Arbeit erledigt wird. Eine Wiederherstellung braucht per Definition eine Kopie. Ein Dienstleister braucht Dateien. Ein Bug, der von realen Datenformen abhängt, braucht etwas, das echt aussieht. Eine Richtlinie, die all das verbietet, wird umgangen, und die Kopien, die dabei entstehen, sind die unverwalteten. Die bessere Frage lautet, welche Kopien gebraucht werden, wer jede verantwortet, wer sie lesen darf und wann sie endet.

Eine zweite Schwierigkeit ist, dass diese Türen von verschiedenen Teams aus verschiedenen Gründen geöffnet werden, sodass kein einzelner Verantwortlicher sie alle sieht. Das Platform-Team besitzt die Backups, das Data-Team die Analytics-Kopie, das Backend-Team Staging, die Finanzabteilung die Beziehung zum Dienstleister, und jeder Entwickler besitzt den Inhalt seines eigenen Laptops. Ein Inventar ist die erste Kontrolle, und man findet es in Storage-Buckets, Job-Zeitplänen und Konfigurationen, nicht in einem Richtliniendokument.

Eine Karte der Seitentüren

Vor den Kontrollen eine Karte. Die Tabelle listet die Türen auf, die dieses Kapitel behandelt, was jede davon in Ledgerlys Fall enthält und wer sie standardmäßig erreicht, wenn niemand etwas anderes entworfen hat.

Tür	Was sie enthält	Wer sie standardmäßig erreicht	Erste Kontrolle
Backups und Volume-Snapshots	Alles, auch Restricted-Spalten und die Secrets-Tabellen	Jeder mit Zugriff auf Storage oder Infrastruktur	Befugnisse zum Erstellen, Wiederherstellen und Löschen trennen; Verschlüsselung mit eigenem Schlüssel
Dumps in gemeinsamem Speicher	Alles, als lesbare Datei	Jeder, der den Ordner öffnen kann	Finden, Verantwortlichen zuweisen, löschen oder nach den Backup-Regeln verschieben
Exporte durch Menschen (CSV, eingefügte Tabellen)	Was auch immer die Abfrage zurückgab	Die Person, dann alle, an die sie es schickt	Abgestufte Genehmigung, Ablaufdatum, keine Restricted-Daten
Staging- und Testdatenbanken	Eine Kopie der Produktion oder ein Abbild	Jeder Entwickler, CI, Auftragnehmer	Maskierte oder synthetische Daten, ausgehende Nachrichten aus
Laptops	Ergebnismengen, lokale Datenbanken, gespeicherte Dateien	Die Nutzer des Laptops und seine Backups	Den Bedarf beseitigen; stattdessen maskierte Stichproben
BI-Tools	Zwischengespeicherte Extrakte, geplante Berichte, geteilte Dashboards	Alle, mit denen das Tool geteilt wird	Nur die Analytics-Kopie lesen, wo möglich Identität pro Person

Tür	Was sie enthält	Wer sie standardmäßig erreicht	Erste Kontrolle
Dateien für Dienstleister	Rechnungs- und Ledger-Exporte	Die Mitarbeiter und Systeme des Dienstleisters	Enge Rolle, Spaltenliste, Aufbewahrung, Ablauf der Datei

Jede Zeile hat dieselbe Form des Problems: eine Datenkopie mit den Kontrollen eines Dateisystems oder eines SaaS-Produkts und eine Leserliste, die niemand aufgeschrieben hat. Das Register aus Kapitel 6 gibt es aus genau diesem Grund bereits, und ich würde es auf jede Zeile oben ausdehnen. Eine Tür, die nicht im Register steht, ist keine Tür, die jemand prüfen wird.

```
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
```

Die neuen Felder sind `storage` und `deleted_by`; `kind` erhält die Werte `backup`, `laptop`, `bi_extract` und `vendor_file`, und `contains` hält die höchste vorhandene Datenklasse fest. Das erste neue Feld sagt, wo man nachsehen muss, wenn jemand fragt, was die Kopie lesen kann, denn die Antwort ist meist eine Storage-Berechtigung und keine Datenbankrolle. Das zweite schließt den Kreis: Ein Datum in `ends`, auf das niemand reagiert, ist ein Wunsch, und die Aufzeichnung darüber, wer die Kopie gelöscht hat, ist der Beleg dafür, dass das Datum etwas bedeutete.

Backups und Snapshots

Ein Backup ist die einzige Seitentür, der niemand widerspricht, und diese Einigkeit ist ein Grund, warum es so oft unkontrolliert bleibt. Es muss alles enthalten, denn seine Aufgabe ist, alles wiederherzustellen. Das ist der gesamte Entwurf, und es ist auch die gesamte Angriffsfläche.

Was ein Backup enthält

Backups gibt es in einigen Arten. Ein **physisches Backup** oder Volume-Snapshot kopiert die Dateien der Datenbank, oft zusammen mit einem fortlaufenden Archiv der Änderungen, damit die Datenbank auf einen Zeitpunkt zurückgesetzt werden kann. Ein **logischer Dump** liest die Daten über die Datenbank und schreibt sie als Anweisungen oder Zeilen. Beide enthalten jede Tabelle. Die physische Art enthält zudem die Rollendefinitionen des Clusters und das Credential-Material, das die Engine dafür speichert, und manche logischen Tools tun das je nach Verwendung ebenfalls.

Daraus folgen zwei Konsequenzen, und keine ist in einem Diagramm des Primary sichtbar. Die erste ist, dass die Restricted-Spalten im Backup liegen, ebenso die Secrets-Tabellen auf Engines, die sie verwenden (`bank_account_secrets` und `payment_token_values`, aus Kapitel 3). Die Regel des Rollenschemas, dass nur die Anwendungsrollen und `human_breakglass` diese Werte erreichen, beschreibt die Datenbank. Über die Datei sagt sie nichts.

Die zweite ist, dass das Backup den Zustand des Tages enthält, an dem es erstellt wurde. Ein Backup aus der Zeit vor der Stilllegung des gemeinsamen Admin-Logins enthält diesen Login noch. Es enthält die Rollenmitgliedschaften von Leuten, die inzwischen das Team gewechselt haben. Rotation und Entzug finden im Live-System statt, und sie reichen nicht in Dateien zurück, die bereits existieren. Wenn jemand eine Rolle auf dem Primary einschränkt, geschieht mit der Wiederherstellung vom letzten Jahr nichts: Der Primary wird besser, und die Kopien behalten den Zustand des Tages, an dem sie entstanden.

Wiederherstellung ist Lesen

Ich würde einen Satz an den Anfang des Backup-Entwurfs stellen: Die Befugnis zur Wiederherstellung ist die Befugnis, alles zu lesen. Wer ein Backup an einem Ort wiederherstellen kann, den er kontrolliert, kann sich dort als Eigentümer anmelden und jede Spalte lesen, und kein Gateway, kein Spalten-Grant und keine Maskierungsregel der Produktion greift, denn die Produktion ist nicht der Ort, an dem gelesen wird.

Das ändert, wem die Kontrolle gehört. Backup-Zugriff wird meist über Infrastrukturrollen vergeben: Jemand kann den Bucket lesen, den Snapshot einhängen oder das Restore-Tool ausführen. Diese Rollen werden selten mit derselben Sorgfalt geprüft wie `human_readonly`, obwohl sie darüber stehen. Die Frage für ein Zugriffs-Review lautet deshalb nicht nur „wer kann den Primary abfragen“, sondern „wer kann wiederherstellen, und wohin“.

Ich würde drei Befugnisse trennen und verschiedenen Principals geben:

Befugnis	Was sie bedeutet	Wer sie hält
Erstellen	Backups erzeugen und in den Speicher schreiben	Die eigene Identität des Backup-Jobs, nur schreibend auf den Speicher
Wiederherstellen	Ein Backup lesen und irgendwo hochfahren	Das Platform-Team, pro Antrag, mit Genehmigung durch eine zweite Person (Vier-Augen-Prinzip)
Löschen	Backups entfernen oder ablaufen lassen	Der Aufbewahrungsprozess; Menschen nur über einen geprüften Antrag

Der Sinn der Trennung ist, dass der Job, der Backups schreibt, sie nicht zurücklesen kann, und dass niemand zugleich ein Backup wiederherstellen und die Wiederherstellungspunkte löschen kann, sodass auf eine Wiederherstellung nicht das Entfernen dessen folgen kann, woraus wiederhergestellt wurde. Wo der Speicher eine Write-once-Aufbewahrung für einen Zeitraum unterstützt, würde ich sie aus demselben Grund nutzen, mit einem Preis: Ein Backup, das Sie aus Versehen erstellt haben, etwa von einer Datenbank, die kurzzeitig etwas enthielt, das sie nicht enthalten durfte, können Sie bis zum Ende des Zeitraums nicht löschen. Entscheiden Sie im Voraus, wie dieser Fall behandelt wird.

Verschlüsselung und wessen Schlüssel

Backups sollten im Ruhezustand verschlüsselt sein, und die interessante Frage ist, wer den Schlüssel hält. Kann derselbe Administrator den Bucket lesen und den Schlüssel verwenden, schützt die Verschlüsselung vor einer verlorenen Platte und vor nichts sonst. Hält ein anderer Principal als die Storage-Administratoren den Schlüssel, und braucht eine Wiederherstellung beide, dann kann das Storage-Team allein die Daten nicht lesen, und der Schlüsselhalter allein hat nichts zu lesen.

Diese Trennung kostet etwas Bestimmtes. Eine Wiederherstellung braucht nun zwei Beteiligte, und ein Backup, dessen Schlüssel verloren ist, ist kein Backup mehr. Ein Weg zur Schlüsselwiederherstellung muss existieren und getestet sein, was ein weiteres Verfahren ist, das das Platform-Team pflegen muss. Kapitel 12 behandelt die Verschlüsselung auf Anwendungsebene und die Grenzen der Schlüsselverwaltung; der engere Punkt hier ist, dass die Backup-Verschlüsselung ein Kontrollpunkt für die Wiederherstellung ist, und ein Kontrollpunkt lohnt sich nur, wenn er ein anderer ist als der auf dem Speicher.

Verschlüsselung hilft nicht gegen die Person, die wiederherstellen darf. Sie schützt die Datei bei der Übertragung und im Ruhezustand vor Menschen und Systemen, die nicht zum Wiederherstellungspfad gehören. Ich würde das im Entwurfsdokument unmissverständlich sagen, denn „die Backups sind verschlüsselt“ wird oft als „die Backups sind sicher“ gehört, und das sind verschiedene Aussagen.

Ein Wiederherstellungsantrag

Die Wiederherstellung ist der Moment, in dem das Backup zu einer Live-Datenbank wird, also gehört die Kontrolle dorthin. Der Block unten zeigt einen Wiederherstellungsantrag mit ausgeschriebener Reihenfolge der Operationen. Backup-Speicher, Register und Audit-Log sind getrennte Systeme, daher deckt keine Transaktion sie ab.

```
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)
```

Bricht der Ablauf nach `registerCopy` und vor dem Start der Wiederherstellung ab, enthält das Register einen Eintrag für eine Kopie, die nicht existiert, was eine wöchentliche Prüfung findet und entfernt. Bricht er nach dem Start der Wiederherstellung ab, existiert die Kopie und ist unerreichbar, und ihre geplante Löschung entfernt sie; ist sogar das Einplanen fehlgeschlagen, findet sie der Sweeper, der auf `ends` im Register reagiert.

Die umgekehrte Reihenfolge ließe eine Kopie zurück, die existiert und nicht aufgelistet ist, und das ist der Fehler, auf den es ankommt. Der Preis des Entwurfs ist die Verzögerung: eine Genehmigung und ein isoliertes Ziel, bevor jemand eine Zeile sieht. In einem Incident ist diese Verzögerung real, weshalb der Wiederherstellungspfad mit im Voraus arrangierter Genehmigung geübt werden sollte, wie Kapitel 15 es für Incidents behandelt. Soll nur eine Handvoll Zeilen statt einer ganzen Datenbank zurückgeholt werden, geht die Wiederherstellung auf dasselbe isolierte Ziel, die Zeilen werden dort entnommen und kommen als Datenkorrektur nach den Regeln von Kapitel 10 zurück in die Produktion.

Eine wiederhergestellte Kopie beginnt geschlossen

Eine wiederhergestellte Datenbank bringt die Rollen, Mitgliedschaften und das Credential-Material vom Tag des Backups zurück. Ich würde sie als feindlich behandeln, bis die Zugriffsdefinition erneut angewendet wurde. In der Praxis heißt das: ein Wiederherstellungsziel ohne Netzwerkroute von Services und ohne menschliche Logins, und ein erster Schritt, der jeden Login deaktiviert, den das Backup zurückgebracht hat, den Superuser und `ledgerly_access_admin` eingeschlossen, denn jedes kehrt mit dem Credential vom Tag des Backups zurück. Dann wird das Superuser-Credential rotiert, und erst danach wird die Zugriffsdefinition aus Kapitel 3 angewendet: Rollen, Mitgliedschaften und Grants, wie der Code sie jetzt beschreibt. Die Kopie wird danach erreichbar und nicht früher.

Dieser Schritt braucht einen Zugriffsadministrator, und es muss einer sein, der zum Ziel gehört. Das Backup enthält bereits einen Login namens `ledgerly_access_admin`, der mit dem Credential des Tages zurückkam, an dem das Backup erstellt wurde. Die Wiederherstellung ersetzt ihn: Der zurückgekehrte Login wird gelöscht und unter demselben Namen mit einem neuen Credential neu angelegt, das für die Wiederherstellung erstellt und mit ihr entfernt wird. Das aktuelle Credential der Produktion für diese Rolle wird nie in das Ziel kopiert, und das im Backup ist verschwunden, bevor sich irgendetwas verbinden kann, denn wer eines von beiden hält, kann innerhalb der Rollen, die es verwaltet, alles gewähren, und eine Wiederherstellungsübung ist genau der Ort, an dem Credentials in eine Testumgebung eingefügt werden. Wird der eigene Administrator der wiederhergestellten Kopie kompromittiert, ist der Auswirkungsbereich diese Kopie.

Erst danach bekommt die wiederhergestellte Kopie Leser, und die Leser sind die Personen aus dem Wiederherstellungsantrag. Eine Wiederherstellung für eine Übung braucht gar keine menschlichen Leser: Die Übung prüft, dass die Daten hochkommen und die Zählungen stimmen, und dann wird die Kopie gelöscht.

Aufbewahrung und Löschung

Aufbewahrung ist ein Abwägen zwischen zwei Kosten. Bewahren Sie Backups länger auf, können Sie sich von Problemen erholen, die spät bemerkt wurden, um den Preis eines größeren Bergs von Daten, der alles enthält, auch Kundendaten, die der Primary längst gelöscht hat. Bewahren Sie sie kürzer auf, halten Sie weniger, und manche Beschädigung ist älter als Ihr ältestes Backup. Ich kenne die richtige Frist für Ihr Unternehmen nicht. Sie hängt davon ab, wie lange ein stilles Problem üblicherweise bis zur Entdeckung braucht, wie viel ein verlorener Monat an Daten kosten würde und was Ihre Rechts- und Compliance-Verantwortlichen zu gelöschten Daten sagen, die in Backups weiterleben. Datenschutz-Regelwerke wie GDPR oder KVKK fragen tendenziell, was mit den Daten einer Person geschieht, sobald sie aus dem Live-System gelöscht sind, und die Auslegung ist deren Sache, nicht die dieses Buchs. Was die Technik liefern kann, ist Evidenz, keine Compliance: ein Register und eine Löschaufzeichnung, die zeigen, was das System tut.

Ein Entwurf, der sich lohnt, mit ihnen zu besprechen: Backups laufen nach Zeitplan ab, und ein Wiederherstellungsverfahren wendet Löschungen, die seit der Erstellung des Backups aufgezeichnet wurden, erneut an, damit eine wiederhergestellte Kopie keine Daten zurückbringt, die absichtlich entfernt wurden. Das fügt jeder Wiederherstellung einen Schritt hinzu und setzt voraus, dass Löschungen an einem Ort aufgezeichnet sind, den eine Wiederherstellung abfragen kann, was ein weiterer Grund ist, Löschanträge in einer dauerhaften Aufzeichnung zu führen. Der Schritt läuft, nachdem die Zugriffsdefinition angewendet wurde, und bevor die Kopie geöffnet wird.

```
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
```

Übungen erzeugen Kopien

Ein Backup, das nie wiederhergestellt wurde, hat nicht gezeigt, dass es funktioniert. Wiederherstellungsübungen sind der Weg, auf dem Sie lernen, dass die Backups funktionieren, und jede Übung erzeugt eine Datenbank mit Restricted-Daten. Ich würde Übungen über denselben Antragspfad laufen lassen wie eine echte Wiederherstellung, mit demselben isolierten Ziel, demselben Registereintrag und einem Enddatum innerhalb des Tages. Die Übung ist zugleich die Probe für den Incident: Braucht sie eine Abkürzung, um fertig zu werden, wird die Abkürzung um drei Uhr nachts genommen.

> **Warnung:** Ein Backup oder Dump, das Restricted-Spalten enthält und in einem Speicher liegt, den eine große Gruppe lesen kann, ist dieselbe Offenlegung wie ein Dauerzugriff auf diese Spalten, ohne Gateway und ohne Log. Wenn Sie eines finden, schränken Sie zuerst die Storage-Berechtigung ein und sprechen Sie danach über die Aufbewahrung.

Dumps, Exporte und CSV-Dateien

Backups sind die Tür mit einem Zweck und einem Team. Dumps und Exporte sind die Türen, die meist eine einzelne Person unter Zeitdruck öffnet, und sie kommen in zwei Arten, die unterschiedlich behandelt werden sollten: die Exporte, die nach Zeitplan laufen, und die, die eine Person erstellt.

Von Hand erstellte Dumps

In Ledgerlys gemeinsamem Ordner liegen einige ältere Dumps. Jeder entstand, weil jemand für eine Stunde eine Kopie brauchte, und die Kopie blieb. Der Dump einer Person ist ein Backup ohne Aufbewahrung, ohne Verschlüsselungsrichtlinie und ohne Verantwortlichen. Auf Engines, auf denen ein Dump über die Datenbank liest, ist er außerdem ein Beleg dafür, dass jemand einen Login hielt, der jede Tabelle lesen konnte.

Ich würde drei Dinge tun. Erstens: Nehmen Sie den Dump aus der Menge der normalen Tools heraus. Das Access-Gateway bietet keinen Dump der ganzen Datenbank an, die menschlichen Rollen können die Tabellen nicht lesen, die ein Dump bräuchte, und `human_breakglass` ist die einzige menschliche Rolle, die es könnte, mit dem Break-Glass-Logging aus Kapitel 8. Zweitens: Finden Sie die vorhandenen Dumps, und prüfen Sie, bevor Sie einen davon löschen, ob er das einzige Backup von jemandem ist. Das kommt vor: Ein Dump wird zum Backup einer Tabelle, die sonst niemand kopiert hat. Bauen Sie zuerst die echten Backups auf, dann löschen Sie die Dumps. Drittens: Geben Sie der Person, die den Dump brauchte, eine Alternative, die die Aufgabe erfüllt, meist die maskierte Analytics-Kopie oder einen Testdatensatz (siehe den Abschnitt zu Testdaten weiter unten).

Exporte durch Menschen

Exporte durch Menschen sind die häufigsten und am schwersten zu verbietenden. Der Support braucht eine Liste der Rechnungen eines Kunden. Die Finanzabteilung braucht eine Datei der Zahlungen des letzten Monats. Das Data-Team braucht eine Stichprobe. Ich würde sie nach der höchsten Datenklasse in der Datei abstufen, denn eine einzige Regel, die für alle passt, ist entweder zu locker für die gefährlichen oder zu schwer für die harmlosen.

Stufe	Inhalt	Kontrolle
Nur Internal	Aggregate, Zählungen, Statusaufschlüsselungen, kein personenbezogener Inhalt	Selbstbedienung, protokolliert, keine Genehmigung

Stufe	Inhalt	Kontrolle
Enthält Confidential	Zeilen mit Kundenkontakten oder Freitext	Ticket mit Begründung, ein Genehmiger, ein Enddatum für die Datei; die Zeilen stammen aus einer JIT-Session mit `SELECT`, die die Spalten benennt (Kapitel 8), sodass der Genehmiger die Spaltenliste liest, die die Datei enthalten wird
Enthält Restricted	Werte von Bankkonten oder Tokens	Wird nicht exportiert; die Antwort ist eine Anzeigeform oder ein Anwendungspfad (Kapitel 12)

Die Stufe wird am Gateway berechnet, nicht vom Antragsteller erklärt, und sie schlägt geschlossen fehl: Sie ist die höchste Katalogklasse (Kapitel 12) unter den Quellspalten, die die Abfrage liest, nicht unter den Namen der Ergebnisspalten, denn ein Alias oder ein Ausdruck wie `lower(contact_email) AS x` verbirgt die Quelle. Eine Spalte, deren Quelle sich nicht auflösen lässt, über eine View, eine CTE oder eine Funktion, zählt als höchste exportierbare Stufe, und wo sich selbst das nicht entscheiden lässt, wird der Export verweigert. Kapitel 9 behandelt die Grenzen der statischen Analyse. Diese Stufen sind die Veranschaulichung einer Form. Entscheidend ist, dass die Genehmigung am Risiko hängt. Eine Genehmigung, die immer, für alles und innerhalb von Sekunden erteilt wird, ist das Theater, das Kapitel 8 beschreibt. Die Stufe, die einen menschlichen Genehmiger braucht, sollte klein genug sein, dass der Genehmiger den Antrag liest, und ich würde das daran messen, ob jemals jemand Nein sagt.

Das Enddatum ist der Teil, den Teams überspringen. Ein Export ohne Enddatum ist eine Kopie, die ihre Frage überlebt. Das Ticket, das die Datei genehmigt, trägt ein Datum, die Datei geht an einen Ort mit einem Verantwortlichen, und ein Sweeper entfernt sie. Die Regel aus Kapitel 10 für Adhoc-Arbeit deckt den leichtesten Fall ab: Fügen Sie Bezeichner und Zählungen ein, keine Zeilen mit personenbezogenem Inhalt. Kapitel 7 behandelt, was das Gateway bei einer großen Ergebnismenge tun kann, nämlich sie begrenzen und sichtbar machen, nicht verhindern.

Der Export für den Dienstleister

Der externe Buchhaltungsdienstleister verbindet sich nie mit einer Ledgerly-Datenbank. Er erhält Dateien, wodurch der Exportschritt eine Tür ist, die Sie vollständig entwerfen können, denn Sie kontrollieren beide Enden

des Pfads bis zum Posteingang des Dienstleisters. Im Ausgangszustand schreibt der Exportschritt von `reporting-job` diese Dateien mit dem gemeinsamen Admin-Login, sodass die Datei alles enthalten kann, was die Admin-Rolle lesen kann. Im Zielzustand verwendet er `app_vendor_export`, und das ändert, was die Datei enthalten kann, denn eine Datei kann nicht mehr enthalten, als der Login, der sie schrieb, lesen konnte.

```
// 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;
```

Der Grant ist die Kontrolle, nicht die Abfrage. Ein Fehler in der Exportabfrage kann Zeilen auslassen oder wiederholen, aber er kann `contact_email` nicht hinzufügen, weil die Rolle sie nicht lesen kann. `customer_id` ist ein opaker Bezeichner, und der Dienstleister hat keine Tabelle, mit der er ihn verknüpfen könnte. Braucht die Arbeit des Dienstleisters Kundennamen, ist das eine Entscheidung, die jemand trifft und genehmigt, und sie stellt sich nicht ein, weil eine Abfrage erweitert wurde.

Der Exportschritt verbindet sich mit `ledgerly_replica_2`, wie die Verbindungsrichtlinie in Kapitel 6 es vorsieht, und schreibt eine Datei. Die Datei geht dann über den mit dem Dienstleister vereinbarten Kanal an ihn. Die Schritte betreffen drei getrennte Systeme, daher zeigt der Block unten die Reihenfolge und was jeder Fehlschlag zurücklässt.

```
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)
```

Hier hängt `auditOrQueue` die Aufzeichnung an das Audit-Log an und stellt sie, wenn das Log nicht erreichbar ist, für einen erneuten Versuch in die Warteschlange und benachrichtigt das Platform-Team, wie in Kapitel 7. Ein Ausfall des Logs wird so zu einer sichtbaren, wiederholten Lücke und nicht zu einer stillen, und keine Transaktion umfasst das Log und den Kanal zum Dienstleister.

Das Manifest ist der Teil, den man gern weglässt. Es lässt beide Seiten prüfen, dass die Datei die gemeinte ist, und es gibt einem Zugriffs-Review etwas zu lesen: so viele Zeilen, diese Spalten, dieser Zeitraum. Taucht in einem Manifest eine Spalte auf, die im Manifest des Vormonats nicht stand, ist das eine Änderung, die jemand hätte genehmigen müssen.

Zwei Grenzen bleiben. Sobald die Datei beim Dienstleister angekommen ist, sind ihre Aufbewahrung eine Frage des Vertrags und des Vertrauens: Sie können den Dienstleister bitten, die Löschung zu bestätigen, aber Sie können sie nicht prüfen. Und die Prüfsumme zeigt, dass die Datei auf dem Weg nicht verändert wurde, nicht, dass die Mitarbeiter des Dienstleisters

die einzigen sind, die sie lesen. Für diese beiden würde ich eine Aufbewahrungsklausel in den Vertrag aufnehmen und eine Zeile im Zugriffs-Review, die fragt, wann der Dienstleister die Löschung zuletzt bestätigt hat.

Testdaten: Staging, maskierte Kopien und synthetische Daten

Staging ist der Ort, an dem das Ledgerly-Szenario begann, und es ist die Tür, die am häufigsten aus Freundlichkeit geöffnet wird. Entwickler wollen realistische Daten, und die Produktion ist die einzige Quelle von Daten mit den Formen, die echte Kunden erzeugen. Der Preis ist, dass Staging das Zugriffsmodell eines Testsystems mit den Daten eines Produktivsystems hat.

Drei Arten von Nicht-Produktionsdaten

Art	Realismus	Angriffsfläche	Kosten
Produktionskopie	Am höchsten	Wie die Produktion, mit schwächeren Kontrollen	Billig zu erstellen, teuer zu besitzen
Maskierte Kopie	Hoch, mit einigen verlorenen Formen	Geringer, nicht null	Eine Maskierungs-Pipeline, die gebaut und gepflegt werden muss (Kapitel 12)
Synthetische Daten	Geringer, begrenzt auf das Modellierte	Am geringsten	Ein Generator, der gepflegt werden muss, und Bugs, die nicht modellierte Formen brauchen

Ich würde Produktionsdaten nicht unmaskiert nach Staging kopieren, und der Grund ist nicht abstrakt. Staging hat meist mehr Leser als die Produktion, einen gemeinsamen Login, CI, die Logs speichert, und Integrationen, die so konfiguriert sind, dass sie sich wie die Produktion verhalten. Es verbindet das größte Publikum mit den schwächsten Identitätskontrollen, und es enthält die sensibelsten Daten. Müssen die Daten echt sein, müssen die Kontrollen so stark sein wie die der Produktion, und dann ist Staging die Produktion unter anderem Namen.

Was eine maskierte Kopie weiterhin ist

Maskierung, behandelt in Kapitel 12, ersetzt oder verschleiert Werte und hält die Form brauchbar. Eine maskierte Kopie ist weit besser als eine rohe, und sie ist dennoch eine Datenkopie, aus Gründen, die man nennen sollte, damit niemand „maskiert“ als Synonym für „unbedenklich weiterzugeben“ behandelt.

Erstens deckt die Maskierung die Spalten ab, die Sie aufgelistet haben. Freitext wie `invoice_lines.description` kann personenbezogene Daten enthalten, die die Klassifizierung nie vorhergesehen hat, weshalb das Buch ihn für den schlimmsten Fall klassifiziert, und eine Maske, die ihn überspringt, lässt ihn in der Kopie. Zweitens lassen formerhaltende Masken die verbleibenden Zeilen unterscheidbar: Ein ungewöhnlicher Rechnungsbetrag, ein ungewöhnliches Datum und ein kleiner Kunde können zusammen auf ein einzelnes Unternehmen deuten. Die Werte wurden maskiert, und die Kombination trägt dennoch Information. Drittens lässt konsistente Maskierung, bei der dieselbe Eingabe dieselbe Ausgabe erhält, damit Joins weiter funktionieren, auch zu, dass dieselbe Person über Kopien und Tabellen hinweg wiedererkannt wird.

Nichts davon heißt, dass Maskierung sich nicht lohnt. Es heißt, dass eine maskierte Kopie als Confidential behandelt wird, bis jemand geprüft hat, was übrig ist, dass ihr Publikum kleiner ist als „alle in der Entwicklung“ und dass sie wie jede andere Kopie ein Enddatum hat; die fortlaufende Analytics-Kopie ist die Ausnahme und wird stattdessen an dem Datum in `review_due` geprüft. Das Wort dafür ist **Re-Identifizierung**: die Möglichkeit, aus verbliebenen Feldern abzuleiten, auf wen sich ein Datensatz bezieht. Ich würde dem Team das Risiko in diesen Begriffen beschreiben und nicht in der Sprache von Angriffen. Die Antwort im Entwurf ist ein kleineres Publikum und ein Ablaufdatum, nicht eine längere Liste maskierter Felder.

Synthetische Daten

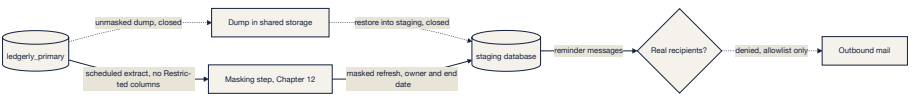
Synthetische Daten enthalten keine echte Person und tragen daher die geringste Angriffsfläche. Ihre Grenze ist, dass sie nur die Formen enthalten, die jemand zu modellieren dachte, und die Bugs, die echte Daten brau-

chen, sind die, die niemand modelliert hat. Ein Generator, der ordentliche Kunden erzeugt, erzeugt nicht den Kunden mit sechs Kontakten und einer Beschreibung in zwei Sprachen.

Ich würde den synthetischen Satz aus den Bugs wachsen lassen. Jedes Mal, wenn ein Problem produktionsförmige Daten zur Reproduktion braucht, wird die Form, die es verursacht hat, und nicht die Werte, zu einer Fixture: ein Kunde mit mehreren Kontakten, eine Rechnungsposition mit leerer Beschreibung, ein Fälligkeitsdatum am Monatsende. Über ein Jahr hört der Satz auf, ordentlich zu sein, und wird zu der nützlichen Art von sonderbar. Er kostet einen Entwickler nach jedem solchen Bug eine Stunde, und er beseitigt den Bedarf an der nächsten Produktionskopie.

Ledgerlys Staging, neu gebaut

Zurück zu Robins Freitag. Der Pfad, der die Tür öffnete, war ein Dump aus dem gemeinsamen Speicher nach Staging. Der Pfad, den ich bauen würde, ersetzt ihn durch einen Refresh, der durch den Maskierungsschritt geht und in einer Datenbank mit kurzer Lebensdauer und benanntem Verantwortlichen landet. Achten Sie im Diagramm auf die gestrichelten Pfade: Das sind die, die der Entwurf schließt.



Die rechte Hälfte des Entwurfs vergisst man leicht. Eine Staging-Umgebung hält Daten und handelt auch auf ihnen. `invoice-worker` in Staging liest Kunden und sendet Erinnerungen, und stimmen seine Einstellungen für ausgehende Nachrichten mit denen der Produktion überein, sendet er sie an die Kontaktadressen in den Daten. Eine maskierte Kopie hat keine echten Adressen, was das nebenbei behebt. Eine rohe Kopie macht aus einem Testlauf Nachrichten an echte Kunden. Ich würde den Ausgangspfad in jeder Nicht-Produktionsumgebung standardmäßig verweigern und nur eine Allowlist interner Testadressen zulassen, unabhängig davon, was die Daten enthalten.

Zahlungs-Tokens folgen einer verwandten Regel. Ein Token steht für ein Zahlungsinstrument, und ob es außerhalb des ausgebenden Systems funktioniert, hängt vom Provider ab, den ich für Ihren nicht beurteilen kann.

Der Standard lautet, dass `payment_tokens.token_value`, `bank_accounts.iban` und `bank_accounts.account_holder_name` in keiner Nicht-Produktionskopie vorhanden sind, maskiert oder nicht (Backups sind die Ausnahme, weil sie alles enthalten müssen), und dass Staging die Test-Tokens des Providers verwendet.

> **Hinweis:** Behandeln Sie den Staging-Refresh als Pipeline mit einem Fehlerpfad, nicht als einmalige Aktion. Schlägt der Maskierungsschritt fehl, hält der Refresh an, und Staging behält seinen bisherigen Inhalt. Ein Refresh, der für einen Lauf auf eine unmaskierte Quelle zurückfällt, erschafft den Dump im gemeinsamen Ordner mit zusätzlichen Schritten neu.

BI-Tools, Laptops und Dienstleister

Drei weitere Türen verdienen eine kurze Behandlung, denn jede holt Daten auf einem Weg aus der Datenbank, der wie eine Funktion aussieht.

BI-Tools. Ein Dashboard-Tool verbindet sich mit einem Login, und alle, die Zugriff auf das Dashboard haben, sehen, was dieser Login lesen kann. Das kehrt das Identitätsmodell aus Kapitel 3 um: Die Datenbank sieht einen Principal, und sagen wir hundert Menschen sehen seine Daten. Zwei Kontrollen helfen. Richten Sie das Tool auf `ledgerly_analytics` und nicht auf ein Replikat, damit es maskierte Daten liest, mit einer eigenen Rolle für das Tool, einer Rolle `app_` wie jede andere Workload (`human_analytics_reader` ist die Rolle des Data-Teams für Menschen), statt mit Produktionsrollen. Und lassen Sie es, wo das Tool es unterstützt, die Identität jeder Person an die Datenbank weitergeben, sodass Zeilen- und Spaltenregeln pro Person gelten. Das Tool führt außerdem eigene Extrakte, Caches und geplante E-Mail-Berichte, die Datenkopien sind und ins Register gehören. Ein Dashboard ist eine bequeme Tür, und ein zwischengespeicherter Extrakt ist dieselbe Tür ohne die Kontrollen.

Laptops. Das Gateway entfernt gespeicherte Credentials und setzt schreibgeschützte Standardwerte, wie Kapitel 7 beschreibt, und es kann nicht kontrollieren, was eine Person mit Zeilen tut, die den Bildschirm erreicht haben. Die praktische Kontrolle ist, die Gründe für eine lokale Kopie zu verringern: eine maskierte Stichprobe, eine Staging-Datenbank, die sich

wie die Produktion verhält, und Fixtures für die Formen, die Bugs auslösen. Ein Team kann nicht beweisen, dass kein Laptop Produktionsdaten enthält, und ich würde das nicht behaupten. Eine Grundausstattung wie Festplattenverschlüsselung und eine Möglichkeit, ein verlorenes Gerät zu löschen, senkt, was ein verlorener Laptop kostet, und sie ist eine Grundausstattung, keine Antwort.

Weitere Dienstleister. Die Buchhaltung ist der, dem das Buch folgt, und andere erhalten Daten als Nebeneffekt: ein Error-Tracker, der einen Request-Body erfasst, ein Helpdesk, das eingefügte Screenshots erhält, eine Log-Plattform, die Anweisungen speichert. Jeder Dienstleister, der eine Kopie hält, bekommt eine Registerzeile mit dem, was sie enthält, und dem Zeitpunkt, an dem der Vertrag endet. Kapitel 12 behandelt, wie sensible Werte aus Logs und Fehlermeldungen herausgehalten werden, was der günstigste Weg ist, sie aus dem Speicher eines Dienstleisters herauszuhalten.

Die Türen bei Ledgerly schließen

Überlegen Sie, wie Alex Ledgerly von verstreuten Kopien zu einem Register mit Verantwortlichen bringen würde. Wie beim Lesepfad in Kapitel 6 kommt es auf die Reihenfolge an, und ich würde sie in sechs Schritten nehmen.

1. **Die Kopien inventarisieren.** Listen Sie Backups, Snapshots, Dumps, Exporte, Staging, BI-Extrakte und Dateien für Dienstleister auf, mit Speicher, Lesern und einem Verantwortlichen. Rechnen Sie damit, Kopien zu finden, an die sich niemand erinnert; der erste Lauf des Registers ist hauptsächlich Entdeckung.
2. **Zuerst den Speicher einschränken.** Schränken Sie bei allem, was Restricted-Spalten enthält, ein, wer den Speicher lesen kann, bevor Sie entscheiden, was gelöscht wird. Das geht schnell und ist umkehrbar, und es schließt die größte Offenlegung, während der Rest sortiert wird.
3. **Die Backup-Befugnisse aufbauen.** Geben Sie dem Backup-Job eine nur schreibende Identität, setzen Sie die Wiederherstellung hinter einen Antrag mit einer zweiten genehmigenden Person, halten Sie den

Verschlüsselungsschlüssel getrennt von den Storage-Administratoren, und führen Sie eine Übung über den Antragspfad aus.

4. **Die Dumps stilllegen.** Sobald die echten Backups existieren und eine Wiederherstellung funktioniert hat, löschen Sie die alten Dumps und halten fest, wer jeden gelöscht hat. Die nächtliche unmaskierte Wiederherstellung, die `ledgerly_analytics` speist, endet, wenn an ihrer Stelle der maskierte Load aus Kapitel 12 unter `app_analytics_load` läuft; bis dahin ist sie die größte offene Tür und bekommt ein Enddatum.
5. **Staging auf maskierte oder synthetische Daten umstellen**, mit standardmäßig ausgeschalteten ausgehenden Nachrichten, und den gemeinsamen Staging-Login entfernen.
6. **Den Export für den Dienstleister auf einen eigenen Login legen.** Stellen Sie den Exportschritt auf `app_vendor_export` um, fügen Sie das Manifest hinzu und legen Sie eine Lebensdauer für Dateien fest.

Verbinden Sie das Register dann mit etwas, das ohnehin geschieht. Das Zugriffs-Review aus Kapitel 14 fragt, welche Kopien existieren und wer jede lesen kann, das Erstellen einer Kopie braucht eine Registerzeile, und ein Sweeper entfernt alles, was nach einer Schonfrist und einer Benachrichtigung des Verantwortlichen über sein `ends` hinaus ist. Wiederherstellungsanträge und Exporte sollten außerdem ins Audit-Log schreiben, das Thema von Kapitel 14. Wiederherstellungs- und Export-Jobs folgen derselben Ausführungsdisziplin wie der Runner, der genehmigte Schreibzugriffe trägt: Sie laufen unter ihrer eigenen Identität, aus einem genehmigten Antrag, mit zuerst geschriebener Aufzeichnung, außer dass ein Export der Stufe Internal die Selbstbedienungsregel aus der Stufentabelle oben behält (Kapitel 9 beschreibt diese Disziplin).

Abwägungen und Kosten

Die erste Kostenart ist die Geschwindigkeit in einem Incident. Eine Wiederherstellung, die einen Genehmiger und ein isoliertes Ziel braucht, ist langsamer als eine, die jeder mit der Bucket-Berechtigung ausführt, und die zusätzliche Zeit fällt auf die diensthabende Person im schlechtesten Moment. Üben Sie den Pfad und arrangieren Sie die Genehmigung im Vor-

aus (Kapitel 15 behandelt den Zugriff während Incidents), damit die Kontrolle ein Verfahren ist, das die Leute kennen, und kein Hindernis, das sie entdecken.

Die zweite Kostenart ist die Schlüsselverwahrung. Den Backup-Schlüssel vom Speicher zu trennen bedeutet, dass ein verlorener Schlüssel ein verlorenes Backup ist, und der Wiederherstellungsweg für den Schlüssel ist ein weiteres Verfahren, das geschrieben und getestet werden muss. Für ein sehr kleines Team kann die Trennung mehr kosten, als sie einbringt; ein verschlüsselter Speicher, enge Storage-Berechtigungen und eine dokumentierte Schlüsselwiederherstellung können ein vernünftiger Anfang sein.

Die dritte Kostenart ist eine langsamere Debugging-Schleife. Wenn Entwickler die Produktion nicht nach Staging wiederherstellen können, dauert die Reproduktion mancher Bugs länger, weil die Form, die sie auslöst, noch nicht in den Testdaten ist. Die Fixture-Gewohnheit zahlt das mit der Zeit zurück und ist am Anfang ein echter Aufwand. Maskierte Kopien fügen eine Pipeline hinzu, die jemand besitzt und die bricht, wenn eine Spalte hinzukommt; deshalb wird in der Migration, die eine Spalte hinzufügt, ihre Klasse entschieden (Kapitel 11).

Die vierte Kostenart ist die Aufbewahrung. Längere Aufbewahrung hält mehr Historie wiederherstellbar und hält mehr sensible Daten an mehr Orten; kürzere Aufbewahrung kehrt beides um. Es gibt keinen Wert, der für alle richtig ist, und ein Team, das keinen gewählt hat, hat „für immer“ gewählt.

Schließlich sind das Register und sein Sweeper Overhead. Ein Sweeper, der eine Kopie löscht, die jemand noch brauchte, ist ein Incident für sich, daher sollte die Löschung nach einer Benachrichtigung und einer Schonfrist erfolgen, und ein Verantwortlicher sollte ein Enddatum einmal mit einer Begründung verlängern können. Ein Register, das nie geöffnet wird, schützt nichts, und die Bindung an das Zugriffs-Review hält es ehrlich.

Wann Sie das alles nicht bauen: Ein kleines Unternehmen mit einer Datenbank, einer Handvoll Entwicklern und ohne Confidential-Daten, die den Namen verdienen, braucht weder abgestufte Exportgenehmigung noch einen geteilten Schlüssel. Es braucht dennoch das Inventar, einen eingeschränkten Speicher für Backups und eine Regel, dass nichts aus der Pro-

duktion unmaskiert in Staging landet. Tun Sie diese drei Dinge, schreiben Sie auf, was Sie zu mehr bewegen würde, und prüfen Sie diese Notiz, wenn sich das Team oder die Daten ändern.

Häufige Fehler

- **Den Zugriff auf den Primary einschränken und seine Backups ignorieren.** Rollen, ein Gateway und ein Audit-Log sind vorhanden, und der Backup-Bucket ist für die ganze Infrastrukturgruppe lesbar. Verlockend ist das, weil Backups „Betrieb“ sind. Die Abhilfe ist, die Wiederherstellung als Lesen von allem zu behandeln, mit eigenem Genehmiger und eigenem Log.
- **Glauben, „verschlüsselt“ heiße „sicher“.** Verschlüsselung im Ruhezustand schützt die Datei vor Menschen außerhalb des Wiederherstellungspfads, und der Schlüssel liegt neben dem Bucket in den Händen desselben Administrators. Halten Sie den Schlüssel getrennt, und sagen Sie, wovor die Verschlüsselung nicht schützt.
- **Alte Dumps löschen, bevor echte Backups existieren.** Der Dump war die einzige Kopie einer Tabelle, und sein Entfernen nahm den einzigen Weg zurück. Bauen und testen Sie zuerst die Backups, dann legen Sie die Dumps still.
- **In ein Netzwerk wiederherstellen, in dem die alten Logins funktionieren.** Die wiederhergestellte Kopie bringt die Rollen und Credentials ihres Tages zurück, auch inzwischen stillgelegte. Stellen Sie in Isolation wieder her, wenden Sie die Zugriffsdefinition erneut an, und ersetzen Sie den Login `ledgerly_access_admin`, den das Backup zurückbrachte, durch ein neues, statt eines der beiden alten Credentials zu behalten.
- **Eine maskierte Kopie als öffentlich behandeln.** Die Werte sind maskiert, und die Kopie wird breit geteilt, dann deutet Freitext oder eine seltene Kombination von Feldern auf einen Kunden. Behandeln Sie sie als Confidential, mit kleinerem Publikum und einem Enddatum, bis sie geprüft ist.
- **Staging als einmalige Ausnahme aus der Produktion auffrischen.** Der Maskierungsschritt war fehlgeschlagen, das Release wartete, und der Dump lag gleich da. Lassen Sie den Refresh geschlossen fehlschla-

gen, sodass die alten Daten bleiben und nichts auf eine rohe Quelle zurückfällt.

- **Ein BI-Tool mit einer Produktionsrolle verbinden lassen.** Ein Login hinter, sagen wir, hundert Betrachtern, mit einem Cache, den niemand aufgelistet hat. Verbinden Sie es mit der Analytics-Kopie über eine eigene Rolle, und registrieren Sie seine Extrakte.
- **Jeden Export in Sekunden genehmigen.** Die Genehmigung existiert und verringert nichts. Reservieren Sie den Genehmiger für die Stufe, die eine Entscheidung verdient, und beobachten Sie, ob jemals jemand ablehnt.
- **Einer Kopie einen Verantwortlichen geben und kein Ende.** Das Register ist vollständig, und nichts verlässt es. Jede Zeile braucht ein Datum in `ends` und einen Sweeper, der darauf reagiert, mit vorheriger Benachrichtigung.

Zentrale Erkenntnisse

- Eine Seitentür trägt die Daten ohne die Kontrollen des Systems, aus dem sie stammen, weil Zugriffskontrollen in der Datenbank leben und nicht in der Datei.
- Die Befugnis, ein Backup wiederherzustellen, ist die Befugnis, alles darin zu lesen, daher verdient der Backup-Zugriff dasselbe Review wie die privilegierteste menschliche Rolle.
- Erstellen, Wiederherstellen und Löschen von Backups sollten bei verschiedenen Principals liegen, und der Verschlüsselungsschlüssel sollte getrennt von den Storage-Administratoren gehalten werden, mit einem getesteten Wiederherstellungsweg.
- Eine wiederhergestellte Kopie kommt mit den Rollen und Credentials des Tages zurück, an dem sie erstellt wurde, daher beginnt sie isoliert, bekommt die Zugriffsdefinition erneut angewendet und erhält einen eigenen Administrator-Login.
- Ein Export für einen Dienstleister wird durch den Grant von `app_vendor_export` kontrolliert, nicht durch die Abfrage, sodass die Datei keine Spalte enthalten kann, die die Rolle nicht lesen kann, und ein Manifest die Datei prüfbar macht.

- Testdaten sollten maskiert oder synthetisch sein, eine maskierte Kopie wird bis zur Prüfung weiterhin als Confidential behandelt, und Staging sollte ausgehende Nachrichten standardmäßig verweigern und geschlossen fehlschlagen, wenn die Maskierung fehlschlägt.
- Jede Kopie braucht einen Verantwortlichen und ein Enddatum in einem Register, das zusammen mit dem Zugriff geprüft wird, denn eine Tür, die niemand besitzt, schließt niemand.

Kopien, Exporte und Wiederherstellungen haben nun Verantwortliche und ein Enddatum, doch ein Entwurf ist nur so gut wie der Beleg, dass er befolgt wird. Das nächste Kapitel wendet sich der Auditierung zu: wo das Audit-Log liegt, was hineinkommt und wie die Zugriffs-Reviews, die dieses Buch immer wieder erwähnt, tatsächlich ablaufen.

TEIL V

Nachweis und Betrieb

Kapitel 14 — Auditierung und Zurechenbarkeit

Stellen Sie sich Ledgerly an einem Dienstagmorgen vor. Ein Kunde hat gefragt, wer im Unternehmen im vergangenen Monat sein Auszahlungskonto angesehen hat, und der Sicherheitskontakt leitet die Frage an Alex weiter. Das klingt nach einer simplen Abfrage. Alex beginnt mit der Session-Liste des Access-Gateways, die zeigt, dass Robin vierzig Minuten lang eine Break-Glass-Session auf dem Primary hatte. Dann öffnet Alex das Datenbank-Log, das einen Session-Login nennt, aber nicht den Grund der Session. Dann durchsucht Alex `audit_events`, das die fachlichen Aktionen festhält, die die Anwendung ausgeführt hat, und nichts darüber sagt, was eine Person eingegeben hat.

Die drei Quellen stimmen bei den meisten Teilen der Geschichte überein und widersprechen sich im Rest. Die Uhr des Gateways geht zwei Minuten vor gegenüber der Uhr der Datenbank. Ein Session-Login taucht im Datenbank-Log ohne jeden Gateway-Eintrag auf, und niemand kann sagen, ob das ein Service-Login ist, den die Liste vergessen hat, oder eine Verbindung, die über einen Pfad kam, den niemand entworfen hat. Alex kann sagen, was wahrscheinlich geschehen ist, kann es aber nicht belegen — und auch nicht belegen, dass niemand mit den Rechten eines Datenbankadministrators die Nachweise bearbeitet hat.

Dieses Kapitel dient der Säule der Zurechenbarkeit aus der These, und es fügt den letzten Schritt des Zielzustands vor den Incidents hinzu: Zugriffe werden in einen Speicher geschrieben, den das System, auf das zugegriffen wird, nicht umschreiben kann, und nach einem festen Zeitplan überprüft. Im Modell aus fünf Teilen (**Identität, Grant, Pfad, Ausführung, Aufzeichnung**) ist dies das Kapitel über die Aufzeichnung. Die Identität sagt, wer gehandelt hat; die Aufzeichnung ist das, was bleibt, und ob sie die Anweisung mit der Person, dem Antrag und dem Ergebnis verknüpft.

Die Frage lautet hier: Was soll eine einzelne Anweisung hinterlassen, wo soll es liegen, und wie erfahren Sie, dass die Nachweise unvollständig sind, bevor jemand danach fragt?

Das Problem und warum es schwierig ist

Im Ausgangszustand hat die Frage keine Antwort. Das Datenbank-Log nennt eine gemeinsame Rolle und keine Person, und es liegt auf Hosts, die die Administratoren kontrollieren. Die früheren Kapitel haben die Identitätshälfte behoben: ein Login pro Session, ein Antrag pro Grant, ein Runner für genehmigte Schreibvorgänge. Jedes davon hat über denselben Aufruf an die Aufzeichnung angehängt, `auditLogAppend`, ohne zu sagen, wohin die Einträge gehen. Dieses Kapitel beantwortet das.

Die Schwierigkeit besteht darin, dass Ledgerly bereits drei Aufzeichnungen hat und jede etwas anderes weiß. Keine davon ist das Audit-Log, wie das Buch es definiert: die Aufzeichnung darüber, wer was mit welchen Daten getan hat, aufbewahrt dort, wo der Handelnde sie nicht ändern kann.

Aufzeichnung	Was sie weiß	Was sie nicht sagen kann	Wer sie ändern kann
Session-Aufzeichnung des Gateways	Die Person, den Antrag, den Grund, die Anweisung wie eingereicht	Was die Datenbank tatsächlich getan hat; ob es einen anderen Pfad gab	Das Platform-Team über den eigenen Speicher des Gateways
Datenbank-Log	Den Login, die Zeit, den Text der Anweisung, falls protokolliert, Fehler	Die Person hinter dem Login, den Antrag, den Grund	Wer auch immer den Datenbank-Host administriert
`audit_events`	Die fachliche Aktion, den handelnden Principal, das Subjekt	Die Anweisungen, die sie hervorgebracht haben; alles, was die Anwendung nicht geschrieben hat	Niemand per `UPDATE`, aber `ledgerly_owner` kann die Tabelle in einer Migration ändern

Die naheliegende Reaktion ist, das Logging hochzudrehen: jede Anweisung in der Datenbank protokollieren und lange aufbewahren. Das fügt eine vierte Aufzeichnung hinzu und schließt keine der drei Lücken. Ein Log, das das System, auf das zugegriffen wird, über sich selbst schreibt, ist

nur so vertrauenswürdig wie seine Administratoren, und ein Log, das niemand liest, ist Theater. Ein Anweisungs-Log ist außerdem selbst eine Kopie sensibler Daten, weil Menschen Literale in Bedingungen eintippen; Kapitel 12 hat davor gewarnt, und dieses Kapitel verantwortet den Mechanismus.

Die Aufzeichnung existiert, um die Lücke im System zu finden, nicht um die Menschen zu bewerten, die es durchlaufen haben. Wenn ein Alarm auf Robins Session auslöst, lautet die erste Frage, was das Design notwendig gemacht hat, nicht, was Robin falsch gemacht hat.

Eine Anweisung, eine Aufzeichnung

Eine Aufzeichnung ist nur nützlich, wenn sie eine vorab aufgeschriebene Liste von Fragen beantworten kann. Ich würde die Liste zuerst schreiben und die Felder daraus ableiten, denn ein Audit-Speicher, der nach den Log-Zeilen entworfen wird, die das Gateway zufällig erzeugt, beantwortet die Fragen, die diese Log-Zeilen zufällig zulassen.

Das Gateway schreibt pro Anweisung zwei Einträge, weil Kapitel 9 verlangt, dass die Absicht vor der Ausführung der Anweisung und das Ergebnis danach geschrieben wird. Zusammen bilden sie die **Anweisungsaufzeichnung**; die beiden Einträge teilen sich eine `statement_id`, und der zweite füllt die Felder, die zu Beginn unbekannt waren. Eine Form dient für beide und für die Aufzeichnungen, die andere Principals schreiben.

```
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
```

Die Art `event` trägt alles, was keine Anweisung ist: das Öffnen einer Session, einen Grant, einen Widerruf, eine Rotation, eine Wiederherstellung, den Start einer Migration. Das sind die Aufrufe `auditLogAppend` und `auditOrQueue` aus den Kapiteln 3, 4, 7, 8, 10, 11, 12, 13 und 15, und die Spalte `action` unterscheidet einen Grant von einem Widerruf oder einer Eindämmung. Sie nutzen dieselbe Kette und dieselben Identitätsfelder, sodass eine einzige Abfrage eine Person von einem Antrag über einen Grant und die Anweisungen, die ihn genutzt haben, bis zum Widerruf verfolgen kann.

Die Tabelle lohnt sich nur, wenn sie die Fragen beantwortet, die Menschen stellen. Die zweite Liste würde ich neben das Schema kleben.

Frage	Felder, die sie beantworten
Wer hat diese Anweisung ausgeführt?	`principal`, `on_behalf_of`, `login`
Mit welchen Grants und welcher Genehmigung?	`roles`, `request_id`, `approver`, `reason`
Was hat sie angefordert?	`statement_class`, `fingerprint`, `literals_hash`, `tables`, `columns`
Was hat sie getan?	`outcome`, `rows_affected`, `rows_returned`
Hat sie Restricted-Daten berührt?	`columns`, nach Datenklasse klassifiziert, mit `rows_returned`
Wann und wie lange?	`started_at`, `ended_at`
Fehlt in ihrer Umgebung etwas?	`stream`, `sequence`, `previous_hash`, die Absicht ohne Ergebnis
Ist dieser Eintrag so, wie er geschrieben wurde?	`entry_hash`, geprüft durch `verifyChain` weiter unten

Zwei Felder tragen den größten Teil der Last. `principal` und `on_behalf_of` sorgen dafür, dass die Spur nicht an einer Maschine endet, wie Kapitel 9 es für die Automatisierung verlangt. Die `request_id` ist der Join-Schlüssel zu allem, was die Kapitel 8 und 10 aufgezeichnet haben.

Eine Aufzeichnung ohne `request_id` ist trotzdem gültig. Eine routinemäßige Session mit `human_readonly` hat keinen Antrag, und ihre Aufzeichnung sagt das, indem sie das Feld leer lässt. Eine leere `request_id` bei einer Session, die Tripel der Rolle `human_jit_write` oder `human_breakglass` hält, ist ein Befund, und die Prüfungen weiter hinten in diesem Kapitel suchen danach.

Ein Speicher, den das System, auf das zugegriffen wird, nicht umschreiben kann

Der Ort, an dem die Einträge liegen, ist wichtiger als ihr Format. Kapitel 1 hat Observability als eine Aufzeichnung definiert, die dort liegt, wo der Principal sie nicht ändern kann. Die Principals, auf die es hier ankommt, sind die mit der größten Macht: die Datenbankadministratoren, das Platform-Team, das das Gateway betreibt, und `ledgerly_access_admin`, des-

sen drei Logins innerhalb der Rollen, die es administriert, fast alles gewähren können. Wenn einer von ihnen die Aufzeichnung bearbeiten kann, belegt sie nur das, was er zu hinterlassen beschlossen hat.

Ich würde den Speicher mit fünf Eigenschaften aufbauen, in dieser Reihenfolge.

1. **Ein separates Konto.** Der Audit-Speicher liegt in einem Cloud-Konto, Projekt oder einer Host-Gruppe, in dem die Datenbankadministratoren, die Betreiber des Gateways und die drei Logins `ledgerly_access_admin` keine Rechte haben. Jemand muss ihn administrieren; das ist eine kleine Gruppe ohne Rechte eines Datenbankadministrators, und ihre eigene Nutzung des Speichers ist für jemanden außerhalb von ihr sichtbar.
2. **Ein Producer, der nur schreiben kann.** Jeder Producer (das Gateway, der Runner, der Credential-Broker, die Pipeline, der Revoke-Job, das Wiederherstellungs-Tooling) hält sein eigenes Credential, das genau eine Operation aufrufen kann: anhängen. Es kann nicht lesen, ändern oder löschen. Ein geleaktes Producer-Credential kann falsche Einträge hinzufügen, was eine Kette erkennbar macht, und kann wahre Einträge nicht entfernen.
3. **Ein Append-Service, der die Kette hält.** Producer berechnen keine Hashes. Ein kleiner Service innerhalb des Audit-Kontos lädt den Head des Streams, verknüpft den neuen Eintrag und speichert beides. Ein Producer, der seine eigene Verknüpfung berechnete, könnte auch alles danach neu berechnen.
4. **Eine Aufbewahrung, die der Kontoadministrator nicht verkürzen kann.** Wo der Speicher eine Write-once-Aufbewahrungssperre bietet, nutzen Sie sie für die Aufbewahrungsfrist, damit niemand im Konto einen Eintrag vor deren Ende löschen kann. Wo er sie nicht bietet, lautet die ehrliche Aussage, dass der Administrator löschen kann, und die Kette und der Anker weiter unten machen eine Löschung sichtbar.
5. **Ein Anker.** Einmal am Tag sendet der Append-Service den Head-Hash jedes Streams an einen zweiten Ort, der einer anderen Gruppe gehört: das System der Sicherheitskontakte oder einen zweiten Speicher. Ohne Anker könnte eine Person mit voller Kontrolle über den Speicher einen Stream ab einem beliebigen Punkt neu schreiben und jeden Hash da-

nach neu berechnen. Mit Anker weicht der neu geschriebene Head vom Anker ab.

```
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 }
```

Zwei Sessions können im selben Moment an denselben Stream anhängen, deshalb wird der Head innerhalb der Transaktion unter einer Sperre gelesen; ohne sie würden beide denselben Head lesen, dieselbe Sequenznummer schreiben und die Kette verzweigen, und der Verifizierer würde dann eine Manipulation melden, die nie stattgefunden hat. Die Regel lautet: ein Writer pro Stream zur selben Zeit, durchgesetzt von dieser Sperre.

Der Verifizierer ist ein separater Job mit einem Nur-Lese-Credential im selben Konto. Seine Arbeit ist der Teil, der aus einem Speicher einen Nachweis macht:

```
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)
```

Nun die ehrliche Grenze, die ich als eigenen Satz ins Design-Dokument schreiben würde. Dieses Design liefert Nachweis, keine Prävention. Es verhindert nicht, dass eine Anweisung läuft, und es verhindert nicht, dass jemand mit genügend Macht über das Audit-Konto den Speicher zerstört. Es macht eine Veränderung erkennbar und reduziert die Menschen, die sie vornehmen könnten, auf eine kurze, separate Liste. Die Erkennung hängt außerdem davon ab, dass jemand die Ausgabe von `verifyChain` liest; deshalb ist der Verifizierer ein geplanter Job mit einem eigenen Heartbeat, den der nächste Abschnitt behandelt.

Die Kosten sind real. Das Audit-Konto ist ein weiteres System, das verfügbar gehalten werden muss, und die Kapitel 8 und 9 haben seine schlimmste Folge bereits akzeptiert: Ein Grant, eine Break-Glass-Erklärung oder eine Anweisung wird abgelehnt, wenn ihre Absicht nicht geschrieben werden kann. Ein Ausfall des Audit-Speichers blockiert daher privilegierte Arbeit, und zwar in dem Moment, in dem jemand womöglich in einem Notfall wartet. Das ist der Preis einer Aufzeichnung, die nicht vom guten Willen abhängt.

> **Warnung:** Legen Sie den Audit-Speicher nicht in der Datenbank ab, die er beschreibt, auch nicht in einem Schema mit eigenem Eigentümer. Wer diese Datenbank administrieren kann, kann die Einträge ändern, und eine Rolle, die nur `INSERT` darf, ändert daran nichts. Die eine strukturelle Ausnahme, die das Buch bereits aufführt, nämlich dass `ledgerly_owner` `ledger_entries` und `audit_events` in einer Migration ändern kann, ist der Grund, warum `audit_events` die fachliche Aufzeichnung der Anwendung ist und nicht das Audit-Log.

Die Lücken finden

Eine Kette beweist, dass die vorhandenen Einträge nicht verändert wurden. Sie beweist nicht, dass jede Anweisung einen Eintrag erzeugt hat. Die zweite Familie von Prüfungen sucht nach dem, was existieren sollte und nicht existiert. Es gibt vier Arten von Lücken, und jede hat eine andere Bedeutung, deshalb meldet die Prüfung sie getrennt.

Lücken in der Sequenz. Ein Stream ist per Konstruktion lückenlos, also bedeutet eine Lücke, dass der Append-Service einen Eintrag verloren hat oder jemand einen entfernt hat. Das ist der seltenste Befund und der schwerwiegendste.

Absichten ohne Ergebnisse. Das Gateway schreibt eine Absicht vor einer Anweisung und ein Ergebnis danach. Eine Absicht ohne Ergebnis nach der längsten Dauer, die das Timeout einer Anweisung erlaubt, bedeutet, dass das Gateway zwischen den beiden ausgefallen ist, dass das Ergebnis in der Retry-Warteschlange wartet oder dass die Anweisung noch läuft. Nur Letzteres ist harmlos, und es sollte das Timeout nicht überdauern.

Sessions nur auf einer Seite. Jeder Datenbank-Login sollte einen bekannten Ursprung haben. Bei Menschen ist der Ursprung eine Gateway-Session. Bei Services ist es ein Lease vom Credential-Broker aus Kapitel 4. Bei Migrationen ist es ein Run der Pipeline. Ein Login im Datenbank-Log mit keinem der drei ist ein Pfad, den niemand entworfen hat, und eine Gateway-Session, die Anweisungen für sich beansprucht, die die Datenbank nie gesehen hat, ist eine Aufzeichnung, die etwas beschreibt, das nicht geschehen ist.

Stumme Producer. Ein Stream, der aufhört zu schreiben, kann eine ruhige Woche bedeuten, einen toten Producer oder einen Producer, den jemand abgeschaltet hat. Die Prüfung vergleicht jeden Stream mit einer erwarteten Frequenz.

```
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)
```

Das Datenbank-Log erreicht das Audit-Konto über einen Shipping-Agent mit eigenem Nur-Schreib-Credential. Wer Zugriff auf den Host hat, kann diesen Agent stoppen, und dafür gibt es die Prüfung auf stumme Streams: Ein gestoppter Agent hinterlässt einen Stream ohne Einträge, und das löst einen eigenen Befund aus.

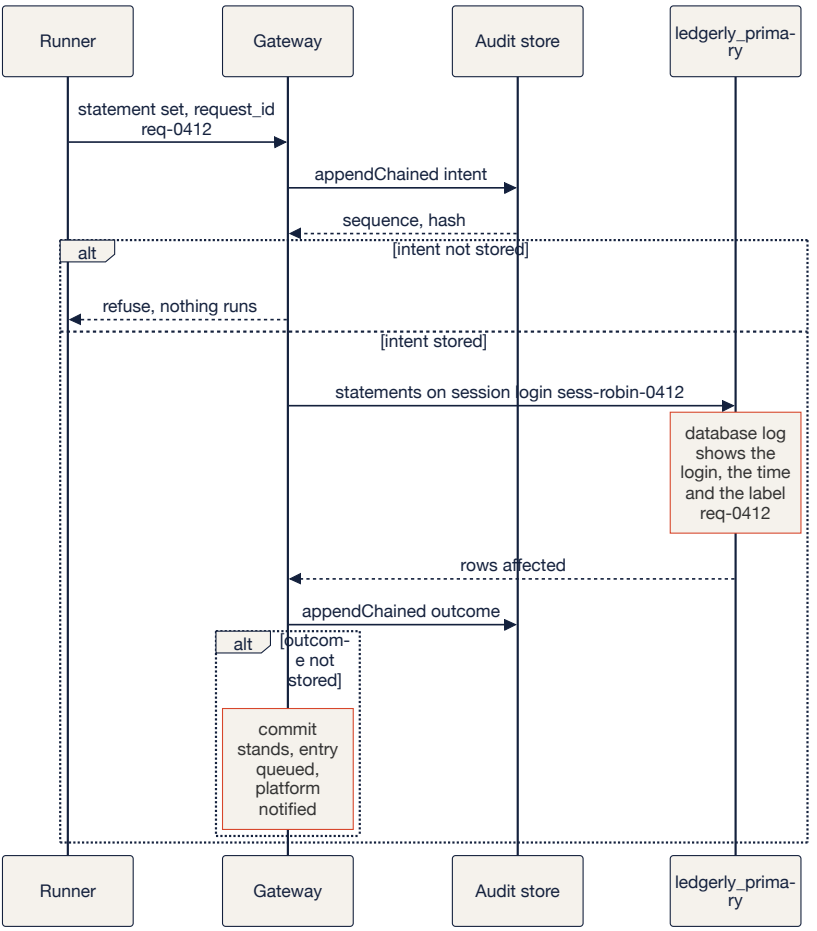
Drei Aufzeichnungen verknüpfen

Die Aufzeichnung des Gateways, das Datenbank-Log und `audit_events` werden von drei Systemen erzeugt, die sich nie eine Transaktion teilen. Sie zu verknüpfen macht aus den Fällen von Alex' Morgen aus Vermutungen eine Zeitleiste, und die Join-Schlüssel müssen vor dem Incident gewählt werden, weil sie sich nachträglich nicht hinzufügen lassen.

Drei Schlüssel leisten die Arbeit. Der **Login** erscheint im Datenbank-Log, und da ein Session-Login genau einer Person oder genau einer Workload-Instanz zugeordnet ist, verknüpft er eine Log-Zeile mit einer Session-Aufzeichnung. Die **Request-ID** wird vom Gateway in die Audit-Aufzeichnung geschrieben und vom Runner in das eigene Feld `request_id` des Audit-

Speichers; sie verknüpft die Anweisung mit der Genehmigung, die sie erlaubt hat. Der Runner schreibt nie `audit_events`, was die Matrix aus Kapitel 9 ihm verbietet. Die Zeilen in `audit_events`, die die in Kapitel 3 eingeführte Correlation-ID tragen, stammen vom Produktpfad, auf dem `admin-console` `payments-service` aufruft, und sie werden über diese ID mit demselben Antrag verknüpft. Das **Connection-Label** aus Kapitel 5 kann bei vermittelten Verbindungen die Request-ID in das Datenbank-Log tragen. Dieses Label setzt, wer die Verbindung öffnet. Bei vermittelter Ausführung öffnet das Gateway sie mit der Request-ID des genehmigten Antrags, daher ist es verlässlich; bei einem Session Handout könnte der Client der Person es auf beliebige Werte setzen, deshalb bleibt der Login der Schlüssel und das Label ein Hinweis.

Das Diagramm zeigt eine genehmigte Datenkorrektur und wo jede Aufzeichnung geschrieben wird; der Runner erreicht den Primary nur über das Gateway. Wenn das Schreiben des Ergebnisses fehlschlägt, bleibt der Commit bestehen, und die Lücke ist erkennbar.



Was die Aufzeichnung nicht enthalten darf, und wer die Auditoren auditiert

Eine Anweisungsaufzeichnung enthält, was Menschen eingegeben haben. Wer einen Kunden anhand der Kontonummer nachschlägt, hat die Kontonummer in die Anweisung geschrieben, und ein Log, das die Anweisung speichert, hat einen Restricted-Wert an einen Ort ohne Schutz auf Spaltenebene kopiert. Die Warnung aus Kapitel 12 gilt uneingeschränkt, also muss der Bauer der Aufzeichnung entscheiden, was aufbewahrt wird.

Ich würde die Anweisung in zwei Teilen speichern. Der **Fingerprint** ist die Anweisung, in der jedes Literal durch einen Platzhalter ersetzt ist, sodass `UPDATE invoices SET status = ? WHERE id IN (?)` die Form festhält und sonst nichts. Der **Literal-Hash** ist ein schlüsselabhängiger Hash der

Literalwerte, der es einer ermittelnden Person erlaubt, einen vermuteten Wert gegen die Aufzeichnung zu prüfen, ohne dass die Aufzeichnung ihn enthält. Der Schlüssel ist wichtig: Literale wie ein Status oder ein kurzes Kontosuffix haben geringe Entropie, und ein Hash ohne Schlüssel ließe sich durch Ausprobieren von Kandidaten umkehren. Der Schlüssel liegt beim Audit-Konto, getrennt von den Producern, und das Werkzeug der ermittelnden Person fordert einen Vergleich an und nie den Schlüssel.

```
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
```

Die letzten Zeilen sind am wichtigsten. Ein Fallback, der die rohe Anweisung speichert, wenn der Parser scheitert, würde genau den Fall aufbewahren, in dem ein Literal am wahrscheinlichsten auftaucht, denn ungewöhnliche Anweisungen sind die, an denen ein Parser scheitert. Ich würde nichts speichern und die Ablehnung für sich sprechen lassen. Der Speicher selbst wird mindestens als Confidential klassifiziert, weil er beschreibt, wer was ansieht, und er hat eine eigene Leserliste und eine eigene Aufbewahrung. Wie viel ein Unternehmen über die eigenen Mitarbeitenden aufzeichnen darf, ist eine Frage für Ihre Rechts- und Compliance-Verantwortlichen.

Wer auditiert `ledgerly_access_admin`

`ledgerly_access_admin` ist die Rolle, die Logins anlegen und innerhalb der Rollen, die sie administriert, alles gewähren kann, genutzt über drei Logins: den Pipeline-Schritt, der die Zugriffsdefinition anwendet (Kapitel 3), den Credential-Broker (Kapitel 4) und das Gateway (Kapitel 7). Kapitel 3 hat gesagt, dass jede Aktion ins Audit-Log geht; der Mechanismus ist, dass jeder der drei ein Producer mit eigenem Stream ist, sodass die Aufzeichnung benennt, welcher Login gehandelt hat. Jedes `CREATE LOGIN`, `GRANT` und `REVOKE`, das er ausführt, ist ein Eintrag der Art `event`, geschrieben vor der Änderung, mit dem Antrag oder Run, der sie verursacht hat.

Das ergibt eine Prüfung, die den Kreis schließt. Der Katalog der Datenbank ist die Wahrheit darüber, welche Grants existieren, und der Audit-Speicher ist die Wahrheit darüber, was gewährt wurde. Ein Grant, der im Katalog ohne passenden Eintrag erscheint, wurde außerhalb der drei Producer erteilt, und das ist der Grant um 2 Uhr nachts, für den die Drift-Prüfung aus Kapitel 3 gebaut wurde. Die beiden Prüfungen sind eine Sache, von zwei Enden gesehen: Die Drift-Prüfung vergleicht den Katalog mit dem Code, und diese hier vergleicht den Katalog mit der Aufzeichnung. Keine repariert etwas; beide melden.

Migrationen

Die Migrations-Pipeline schreibt in den Audit-Speicher, bevor sie eine Migration ausführt, wie Kapitel 11 es verlangt, und noch einmal, wenn sie fertig ist oder scheitert. Der Grund für den externen Eintrag ist, dass das Migrationswerkzeug auch in einer Tabelle innerhalb von `ledgerly_primary` festhält, was es angewendet hat, und `ledgerly_migrator` die Rolle `ledgerly_owner` übernimmt, die diese Tabelle ändern kann. Eine Aufzeichnung, die der Handelnde umschreiben kann, ist eine Bequemlichkeit und kein Nachweis. Die Regel lautet: Eine Migration gilt als angewendet, wenn der externe Starteintrag und der Commit-Hash im Audit-Speicher existieren, und eine Zeile in der Migrationstabelle ohne passenden externen Eintrag wird durch denselben Vergleich von Katalog und Aufzeichnung gemeldet, als eine Änderung, die an der Pipeline vorbeiging.

Alarme, auf die jemand antwortet

Eine Aufzeichnung beantwortet Fragen im Nachhinein. Einige dieser Fragen lohnt es sich, laufend zu stellen, und ein Alarm ist die Form, die das tut. Der Test, den ich auf jede Regel anwenden würde, lautet, ob jemand benannt ist, der darauf antwortet, und weiß, was zu tun ist.

Ledgerly beginnt mit drei Regeln, die jeweils etwas ausnutzen, das die Aufzeichnung weiß und die anderen Quellen nicht.

```
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)
```

Die Schwellenwerte sind Illustrationen; der richtige `nightRowThreshold` ist der, der einer Person eine Handvoll Ereignisse pro Woche vorlegt. Die zweite Regel ist die unabhängige Kontrolle der Erklärung aus Kapitel 8: Jenes Kapitel alarmiert, wenn eine Person den Notfall erklärt, und diese Regel findet eine Session, die die Rolle ohne Erklärung hält, also einen Grant, der daran vorbeiging. Die dritte Regel sollte fast nie auslösen. `human_readonly` kann keine Restricted-Spalten lesen, und ein JIT-Antrag kann keine benennen; ein lesender Zugriff eines Menschen auf Restricted-Daten ist also entweder eine erklärte Break-Glass-Session oder ein Grant, der nicht zur Matrix passt, und Letzteres ist ein Design-Befund.

Jede Regel braucht einen benannten Verantwortlichen und eine erste Aktion: Der Sicherheitskontakt fragt, was die Aufgabe war, die diensthabende Person im Platform-Team bestätigt die Erklärung und eröffnet den Review-Punkt, das Platform-Team widerruft einen umgangenen Grant. Eine Regel, die quittiert wird, ohne dass jemand hinsieht, ist Sicherheitstheater.

Das Zugriffs-Review

Alarme beobachten Ereignisse. Das **Zugriffs-Review** ist die geplante Prüfung, ob der Zugriff jedes Principals noch gebraucht wird, und es ist die Kontrolle, die den schleichenden Ausfall fängt, den kein Ereignis zeigt: Zugriff, der bei der Vergabe richtig war und unbemerkt aufgehört hat, richtig zu sein. Menschen wechseln das Team, ein Service hört auf, eine Tabelle zu lesen, eine Rolle behält ein Privileg, das ihre Workload seit Monaten nicht genutzt hat. Kapitel 3 hat gesagt, dass ein falscher Grant, korrekt deklariert, die Drift-Prüfung besteht und dass das Zugriffs-Review der Ort für dieses Urteil ist.

Das Review vergleicht, was gewährt ist, mit dem, was genutzt wurde. Die Aufzeichnung liefert die zweite Liste, und das macht es zu einer Entscheidung und nicht zu einem Formular zum Unterschreiben.

```
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)
```

Drei Regeln bestimmen, wie ich seine Ausgabe nutzen würde. Erstens schlägt die Funktion nur vor und widerruft nie, aus dem Grund, den Kapitel 3 genannt hat: Ein automatischer Widerruf eines Grants, auf den jemand angewiesen ist, ist eine neue Art, die Produktion lahmzulegen. Zweitens muss der Parameter `period` länger sein als der längste Zyklus der Workload. Ein Grant, den ein Quartalsbericht nutzt, ist in den meisten Monaten ungenutzt, also würde ein Review mit einem Monat in zwei von drei Monaten vorschlagen, ihn zu widerrufen. Drittens wird die Nutzung nach Privileg und nicht nur nach Tabelle abgeglichen. Eine Rolle, die jede Nacht aus einer Tabelle liest und sie nie aktualisiert hat, ist ein Kandidat für die Einschränkung von `S, U` auf `S`, und die Aufzeichnung hat, was dafür nötig ist, weil jede Aufzeichnung die Anweisungsklasse und die berührten Tabellen trägt.

Das Review hat einen zweiten Input, den die Aufzeichnung nicht liefern kann: Dinge, die keine Grants sind. Das Register der Kopien aus Kapitel 13 gehört auf die Tagesordnung, ebenso die Liste, wer JIT-Anträge genehmigen darf, und die Liste, wer den Audit-Speicher lesen kann. Die Frage ist für jedes dieselbe: Braucht die Person oder das System das noch, und ist noch jemand dafür verantwortlich? Der Anhang enthält eine Vorlage.

Nachweis, nicht Compliance

Datenschutz- und Zahlungskarten-Regelwerke stellen tendenziell eine Familie von Fragen zum Zugriff: Wer kann die Daten erreichen, wer hat was getan, wie lange dauert der Zugriff, wo sind Kopien, und was geschieht im Notfall. Ein Auditor, der unter einem dieser Regelwerke arbeitet, will meist sehen, dass eine Kontrolle existiert, dass sie gelaufen ist und dass ihre Ergebnisse angesehen wurden. Ihre Rechts- und Compliance-Verantwortlichen verantworten die Auslegung jedes Regelwerks. Dieser Abschnitt behauptet nicht mehr, als die oben beschriebene Technik liefern kann, nämlich Nachweis: Aufzeichnungen, die Fragen beantworten, und einen Weg zu zeigen, dass sie nicht verändert wurden.

Nichts in diesem Kapitel macht ein Unternehmen mit irgendetwas konform. Die Tabelle listet die Fragen auf, die ein Auditor tendenziell stellt, die Kontrolle, die jede beantwortet, das Artefakt, das sie hinterlässt, und das Kapitel, das sie aufbaut; der Anhang wiederholt sie als Zuordnung.

Frage	Kontrolle	Artefakt	Kapitel
Wer kann Produktionsdaten erreichen, und über welchen Pfad?	Gateway als einzige Tür, Rollenschema als Code, Drift-Prüfung	Liste der Netzwerkregeln, Rollendefinitionen, Drift-Bericht	3, 7
Wer hat eine bestimmte Anweisung ausgeführt, mit welchen Grants und welcher Genehmigung?	`access_audit_record` mit Request-ID und Genehmiger	Die Aufzeichnung und der Bericht von `verifyChain`	14
Wie lange dauert erhöhter Zugriff, und wer hat ihn genehmigt?	JIT-Antrag mit Genehmiger und `expires_at`, Revoke-Job	Antragsdatensätze, Revoke-Log, Prüfung überfälliger Grants	8
Wie werden sensible Daten identifiziert und begrenzt?	Klassifizierung im Schema, Maskierung, zurückgehaltene Spalten	Der Katalog, die Verifikation des maskierten Ladens	12
Wo sind Kopien von Produktionsdaten, und wer verantwortet jede?	Register mit Verantwortlichem und Enddatum, Sweeper	Register, Sweeper-Log	6, 13
Wer ändert das Schema, und wer hat es geprüft?	Migrations-Pipeline, geschützter Branch, externer Starteintrag	Migrationsdatensätze, abgeglichen mit Commits und Audit-Einträgen	11
Wird der Zugriff noch gebraucht?	Zugriffs-Review mit benannten Reviewern	Abgeschlossenes Review mit Entscheidungen	14
Was geschieht im Notfall?	Break-Glass mit Alarm und verpflichtendem Review	Review-Datensätze, Alarmhistorie	8, 15
Wer administriert den Zugriff selbst?	Die Logins `ledgerly_access_admin` als separate Producer	Ihre Streams und der Vergleich von Katalog und Aufzeichnung	3, 14

Abwägungen und Kosten

Die Aufzeichnung hat eine Rechnung, und der größte Teil wird an zwei Stellen bezahlt.

Speicher und Aufmerksamkeit. Eine Aufzeichnung pro Anweisung, zweimal, wächst mit dem Verkehr, und der Speicher braucht eigene Aufbewahrung, eigene Backups und eine eigene Leserliste. Die größeren Kosten sind

die Aufmerksamkeit. Ein Speicher, den niemand abfragt, ist teures Sicherheitstheater, daher gehören die wöchentliche Stichprobe erhöhter Sessions aus Kapitel 7, die Alarme und das Review zum Preis und sind keine Extras.

Eine neue Abhängigkeit auf dem kritischen Pfad. Weil die Absicht vor der Ausführung der Anweisung geschrieben wird, blockiert ein Ausfall des Audits privilegierte Arbeit. Die Alternative, Anweisungen auszuführen und später zu protokollieren, macht die Aufzeichnung genau dann optional, wenn sie zählt. Ich würde die Ablehnung für Grants, Break-Glass und Schreibvorgänge beibehalten und für routinemäßige Lesezugriffe mit `human_readonly` einen Schreibvorgang über eine Warteschlange erwägen, mit einer Obergrenze dafür, wie lange die Warteschlange wachsen darf, falls das Team entscheidet, dass ein Ausfall bei Lesezugriffen mehr kostet als eine Lücke. Das ist eine Entscheidung, die das Platform-Team einmal treffen und aufschreiben sollte.

Wann man nicht alles bauen sollte: Ein Unternehmen mit einer Handvoll Entwicklern, einer Datenbank und noch ohne Gateway sollte nicht mit einem Append-Service, einem Anker und einem separaten Konto beginnen. Beginnen Sie mit drei Dingen: die Session jeder Person unter ihrem eigenen Login protokollieren, das Datenbank-Log vom Host an einen Ort schicken, aus dem die Administratoren nicht löschen können, und eine benannte Person auf ein monatliches Review ansetzen. Fügen Sie die Kette und den Anker hinzu, wenn die Zahl der Menschen, die das Log verändern könnten, „Vertrauen“ zu einer unzureichenden Antwort macht.

Häufige Fehler

- **Das Datenbank-Log als Audit-Log behandeln.** Es ist die vollständigste Quelle, und es liegt unter der Kontrolle der Administratoren. Schicken Sie es woandershin und nutzen Sie es als einen Input für den Join, nicht als die Aufzeichnung.
- **Eine Hash-Kette ohne separaten Speicher.** Die Kette sieht aus wie ein Beweis und lässt sich von jedem neu berechnen, der in den Speicher schreiben kann. Legen Sie den Speicher in ein anderes Konto, halten Sie die Kette im Append-Service und verankern Sie den Head.
- **Die rohe Anweisung speichern.** Es ist verlockend, weil es die Ermittlung einfach macht, und es kopiert Restricted-Werte in einen Speicher

ohne Spaltenschutz. Speichern Sie den Fingerprint und einen schlüsselabhängigen Hash, und verweigern Sie den Fallback auf Rohtext.

- **Alarme ohne Verantwortlichen.** Ein Kanal, den niemand zu beantworten hat, wird zu Rauschen. Geben Sie jeder Regel einen Verantwortlichen und eine erste Aktion.
- **Alles auditieren außer den Auditoren.** Die Rolle, die Zugriff gewährt, ist die, deren Handlungen am meisten zählen. Geben Sie ihr eigene Producer und vergleichen Sie ihre Arbeit mit dem Katalog.

Zentrale Erkenntnisse

- Das Audit-Log ist die Aufzeichnung darüber, wer was mit welchen Daten getan hat, aufbewahrt dort, wo der Handelnde sie nicht ändern kann, und die Aufzeichnung des Gateways, das Datenbank-Log und `audit_events` sind drei Inputs dafür, nicht die Sache selbst.
- Eine Anweisungsaufzeichnung entsteht aus einer Liste von Fragen, die zuerst geschrieben wird, und ihre Identitätsfelder enden über `principal` und `on_behalf_of` bei einer Person, auch wenn eine Maschine gehandelt hat.
- Manipulationsnachweis entsteht durch ein separates Konto, Producer, die nur schreiben können, einen Append-Service, der die Kette hält, und einen Anker, der jemand anderem gehört, und er liefert Nachweis und nie Prävention.
- Eine Kette zeigt, dass vorhandene Einträge nicht verändert wurden; deshalb suchen separate Prüfungen nach Lücken in der Sequenz, Absichten ohne Ergebnisse, Datenbank-Logins ohne bekannten Ursprung und Producern, die stumm geworden sind; die drei Aufzeichnungen werden über den Login und die Request-ID verknüpft, und das Connection-Label ist nur ein Hinweis.
- Die Aufzeichnung darf nie zu einer Kopie der Daten werden, die sie beschreibt; deshalb werden Anweisungen als Fingerprints mit einem schlüsselabhängigen Hash der Literale gespeichert, und der Speicher wird klassifiziert und beschränkt wie die Daten.
- Alarme brauchen einen Verantwortlichen und eine erste Aktion, und das Zugriffs-Review vergleicht Grants mit der Nutzung über einen

Zeitraum, der länger ist als der Zyklus der Workload, damit Widerrufe aus Nachweisen vorgeschlagen werden.

- Nachweis ist nicht Compliance: Die Zuordnung von Frage, Kontrolle, Artefakt und Kapitel zeigt, was das System belegen kann, und Ihre Rechts- und Compliance-Verantwortlichen verantworten, was dieser Beleg unter einem Regelwerk bedeutet.

Die Aufzeichnung existiert nun, ist verknüpft und wird beobachtet. Es bleibt der Tag, an dem sie unter Druck gebraucht wird, wenn die fragende Person kein Auditor mehr ist, sondern eine diensthabende Person mit einem geleakten Credential. Das nächste Kapitel wendet sich dem Zugriff bei Incidents zu und nutzt die hier aufgebaute Aufzeichnung, um herauszufinden, was ein kompromittierter Principal tatsächlich berührt hat.

Kapitel 15 — Zugriff bei Incidents

Stellen Sie sich einen Samstagnachmittag bei Ledgerly vor. Sam aus dem Support schreibt in den Kanal der Rufbereitschaft: Ein Firmen-Laptop wurde aus einem geparkten Auto gestohlen, und auf ihm war möglicherweise eine Gateway-Session offen. Robin hat Rufbereitschaft und liest die Nachricht zweimal. Es ist nicht bekannt, dass etwas nicht stimmt, aber „möglicherweise offen“ ist die Art von Satz, die beantwortet und nicht abgewogen werden muss.

Robin braucht drei Dinge, und die Reihenfolge zählt. Das Erste ist, alles zu stoppen, was diese Session und jeder Login dahinter noch tun kann. Das Zweite ist herauszufinden, was sie getan hat, solange sie es konnte. Das Dritte ist, den Betrieb am Laufen zu halten, während die ersten beiden geschehen — das heißt, andere Personen brauchen weiterhin die Datenbank, einige davon dringend, und die üblichen Genehmiger sind zu Hause.

Dieses Kapitel behandelt den Zugriff bei Incidents als etwas, das man entwirft und übt. Es beansprucht alle drei Säulen gleichzeitig: die Autorisierung, wenn Grants in Eile entfernt oder hinzugefügt werden, die Ausführung, wenn Korrekturen unter Druck laufen müssen, und die Zurechenbarkeit, wenn die Aufzeichnung sagen muss, was geschehen ist. Im Modell aus fünf Teilen findet die Identität die Logins, Grant und Pfad trennen sie ab, die Ausführung trägt die Notfall-Schreibvorgänge, und die Aufzeichnung misst den Schaden. Dieses Kapitel fügt den letzten Schritt des Zielzustands von Ledgerly hinzu: Der Incident-Zugriff hat einen geübten Pfad, und der Widerruf ist Routine. Die Frage lautet, wie Ledgerly den Zugriff stoppt, misst, was erreicht wurde, und weiterarbeitet, ohne die Türen wieder zu öffnen, die die früheren Kapitel geschlossen haben.

Das Problem und warum es schwierig ist

Ein Incident ist der Moment, in dem jede frühere Kontrolle wie ein Hindernis aussieht. Die Genehmigung, die an einem ruhigen Dienstag vernünftig war, ist an einem Samstag eine Wartezeit, das Gateway, das alles aufzeichnet, ist eine weitere Komponente, die ausfallen kann, und die sauberste Abkürzung im Raum ist ein Login, der all das überspringt. Die übliche Reaktion ist, diesen Login „für Notfälle“ im Voraus zu bauen. Es ist wieder das gemeinsame Admin-Passwort, nur mit einem besseren Namen.

Drei Dinge sind unter Druck anders. Die Zeit ist knapp, also werden Schritte übersprungen, an die man sich erst erinnern muss. Das Wissen ist unvollständig, also liegt die erste Schätzung des Umfangs in der einen oder anderen Richtung daneben. Und die Handelnden sind ungewöhnlich: Menschen, die selten an die Datenbank gehen, werden plötzlich darum gebeten, und Menschen, die täglich an ihr arbeiten, sind müde.

Die Entwurfsfrage lautet daher, welche Kontrollen sich lockern und welche nicht. Ich würde eine Regel festhalten: Dringlichkeit verändert die Geschwindigkeit, nie die Identität, nie die Anweisung, die läuft, und nie die Aufzeichnung. Wartefristen und die Reihenfolge der Genehmiger dürfen schrumpfen. Ein Principal, der unter seinem eigenen Login handelt, eine an einen Anweisungs-Hash gebundene Genehmigung und ein Eintrag im Audit-Log dürfen es nicht. Wenn sich eine Kontrolle nicht verkürzen lässt, ohne eines dieser drei zu entfernen, ist die Kontrolle falsch, nicht die Regel.

Die Geschwindigkeit muss gebaut werden. Ein Widerruf, der ein Dutzend manueller Schritte braucht, wird in der falschen Reihenfolge oder nur halb erledigt. Die Reihenfolge ist Code, die Abfrage des Umfangs ist Code, und der Abgleich, der sagt, ob der Incident Zugriff hinterlassen hat, ist Code. Ein Runbook, das darauf verweist, ist ein Verfahren; eines, das sie durch Urteilsvermögen ersetzt, ist eine Hoffnung.

Eine Kompromittierung eindämmen

Eindämmung bedeutet hier, den Zugriff eines einzelnen Principals zu stoppen, ob die Ursache nun ein verlorener Laptop, ein geleaktes Credential oder eine Workload ist, die sich auf eine Weise verhält, die niemand erwartet hat. Sie entscheidet nicht über die Ursache. Kapitel 4 hat zwei

Fragen getrennt, die Panik gern vermischt: wie man ein Credential am Funktionieren hindert, und was es getan hat, solange es funktionierte. Die Eindämmung beantwortet die erste, und die Reihenfolge unten hält die zweite beantwortbar.

Die Reihenfolge lautet: Grants widerrufen, Sessions schließen, Credentials rotieren, den Schaden anhand des Audit-Logs eingrenzen, dann aufzeichnen und benachrichtigen. Jeder Schritt steht aus einem Grund an seiner Stelle.

Das Widerrufen von Grants kommt zuerst, weil eine Session ohne Privilegien nichts tun kann und ein erneuter Verbindungsaufbau nichts erhält. Das Schließen der Sessions folgt an zweiter Stelle, für die laufende Arbeit und die offenen Verbindungen, die einen widerrufenen Grant überdauern (Kapitel 4 hat dasselbe zu abgelaufenen Credentials gesagt). Die Rotation steht an dritter Stelle, weil sie neue Logins stoppt und für die bereits offenen nichts tut. Die Eingrenzung ist rein lesend und braucht die Aufzeichnung, nicht den Zugriff, deshalb wartet sie, bis der Zugriff weg ist. Die Benachrichtigung kommt zuletzt, die Aufzeichnung jedes Schritts aber nicht: Jeder Schritt wird angehängt, sobald er beendet ist.

```
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
```

Zwei Details verdienen eine Anmerkung. Die Funktion gibt die unfertigen Schritte zurück und wirft keine Ausnahme, denn ein halb eingedämmter Principal ist ein Zustand, den man verfolgen muss, und eine Ausnahme würde den einen Prozess beenden, der davon weiß. Und `auditOrQueue` ist derselbe Helfer wie in Kapitel 7: Ein Ausfall des Logs verzögert den Eintrag und nie das Entfernen des Zugriffs. Die Asymmetrie ist dieselbe, die Kapitel 8 für den Widerruf gewählt hat: Ein Grant ohne Log-Eintrag ist eine Gefährdung, ein Widerruf ohne Eintrag eine Lücke, die sich später füllen lässt.

Was jeder Schritt bedeutet, hängt davon ab, was der Principal ist. Die Tabelle listet die vier Fälle auf, die Ledgerly hat.

Principal	Grants widerrufen	Sessions schließen	Credentials rotieren
Person, Session-Logins	Die JIT-Tripel, `human_breakglass` und `human_readonly` von jedem Session-Login entfernen, den die Person hält	`endSession` für jeden Login; die Person im Identity Provider deaktivieren, damit keine neue Session geöffnet wird (Kapitel 7)	Die Credentials und Faktoren der Person im Identity Provider zurücksetzen; Session-Logins werden verworfen, es gibt also kein Datenbankpasswort zu rotieren
Service, dynamisches Credential	Den Lease beenden; der Login geht damit (Kapitel 4)	Der Broker schließt die Sessions als Teil der Beendigung des Leases	Nichts von Hand; die nächste Anfrage erhält einen neuen Wert
Service, statisches Credential in einem A/B-Paar	Die Rolle `app_*` sofort vom geleakten Login entfernen, ohne darauf zu warten, dass der alte Login ausläuft	Seine verbleibenden Sessions schließen	Einen neuen Wert für den ruhenden Login veröffentlichen und den Zeiger umschalten; Services neu starten, die ihr Secret nicht neu lesen können
Login von `ledgerly_access_admin`	Seine Privilegien entfernen, was die Bereitstellung für den jeweiligen Principal stoppt, der ihn genutzt hat	Seine Sessions schließen	Diesem Principal ein neues Credential ausstellen; die Frage nach dem Umfang wird zu der, welche Grants und Logins kein Antrag erklärt (die Drift-Prüfung aus Kapitel 3)

Die Zeile zum statischen Credential entspricht Kapitel 4. Ein geleaktes statisches Credential ist keine geplante Rotation, und der geleakte Login wird getrennt, bevor sein Ersatz bereit ist, was in den Services, die ihn nutzen,

einen Ausfall kostet. Kapitel 4 hat diese Kosten bereits akzeptiert, und ich würde die Akzeptanz beibehalten: Ein geleaktes Credential, das weiter funktioniert, ist das größere Problem. Die Services, die einen Neustart brauchen, sollten auf einer Liste stehen, bevor der Tag kommt, an dem es zählt. Im Ausgangszustand, in dem sich alle Services einen Admin-Login teilen, ist Schritt eins der Ausfall des ganzen Unternehmens — das stärkste Argument dafür, den routinemäßigen Widerruf zuerst zu bauen.

Die letzte Zeile ist die schwere. Der Login, der den Zugriff administriert, kann innerhalb der Rollen, die er administriert, alles gewähren; ihn einzudämmen hindert also das Gateway, die Pipeline oder den Broker daran, irgendetwas bereitzustellen, bis ein neues Credential existiert. Das ist der Preis dafür, den Login mit der größten Reichweite zu trennen, und ein Team sollte im Voraus entscheiden, wer ihn zahlen darf.

Den Auswirkungsbereich aus der Aufzeichnung eingrenzen

Sobald der Zugriff weg ist, wird die Frage zu der, was er getan hat. Die Antwort ist eine einzige Abfrage über `access_audit_record`, dessen Entwurf Kapitel 14 beschreibt, ausgeführt mit den Feldern, die jenes Kapitel definiert. Die Abfrage liefert den Auswirkungsbereich eines Principals seit dem Zeitpunkt T, zu dem sein Zugriff erstmals missbraucht worden sein könnte: jede Tabelle, die er berührt hat, wie viele Zeilen zurückkamen, wie viele geändert wurden und welche dieser Tabellen Restricted-Daten enthalten.

```
// 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;
```

Bei der Wahl von `:logins` und `:since` steckt der größte Teil des Urteilsvermögens. Bei einer Person erzeugt das Gateway für jede Session einen neuen Login, die Menge besteht also aus jedem Session-Login, den der Principal seit T hatte, nicht aus einem. Für T nehmen Sie den frühesten Moment, in dem der Zugriff missbraucht worden sein könnte, etwa den Zeitpunkt, an dem der Laptop zuletzt nachweislich in den Händen seines Besitzers war, und nicht den Zeitpunkt, zu dem es jemand bemerkt hat. Ein zu enges Fenster liefert eine saubere Antwort auf die falsche Frage.

Das Ergebnis ist nur so gut wie die Aufzeichnung, deshalb kommen drei Prüfungen, bevor es jemand liest.

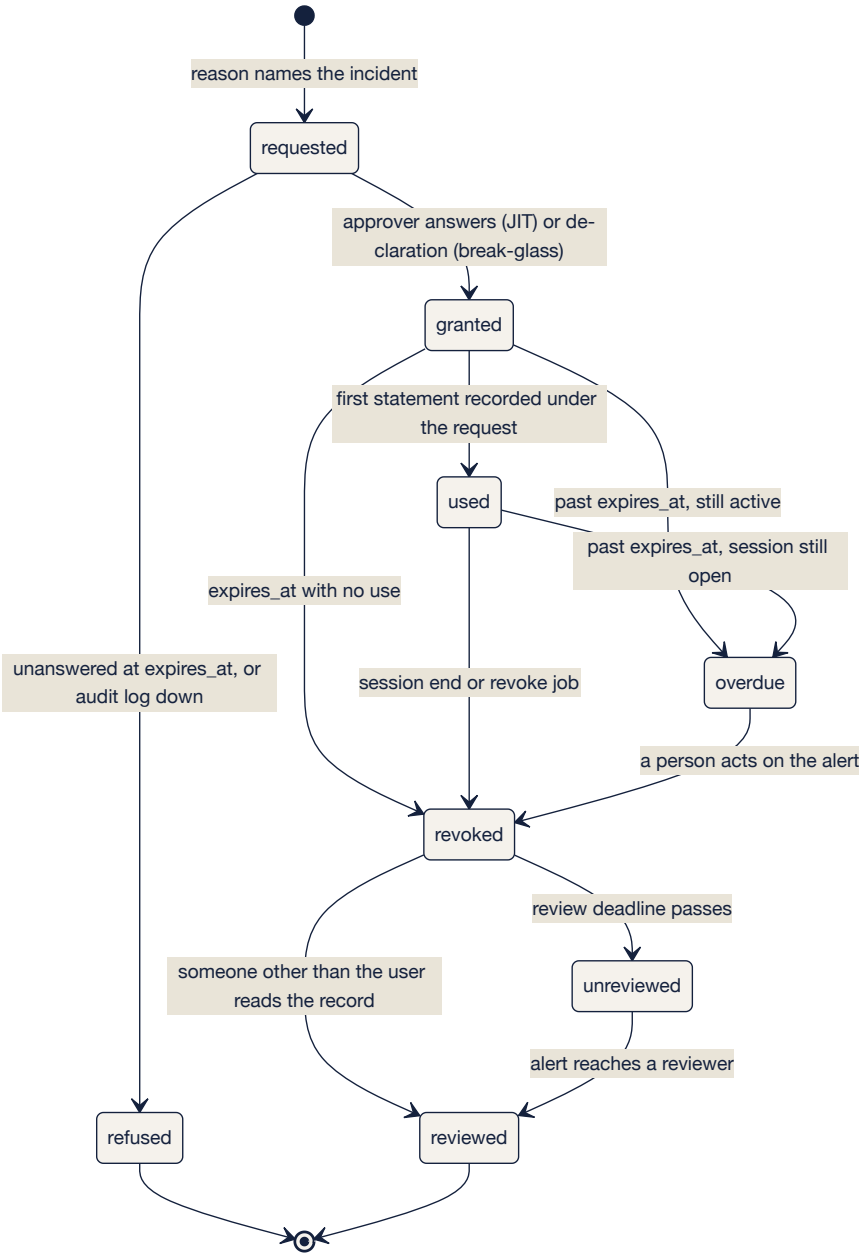
1. `verifyChain` besteht über das Fenster, wie Kapitel 14 es definiert. Schlägt es fehl, ist das Ergebnis ein Nachweis für etwas anderes als den Incident, und der Incident bekommt eine zweite Aufzeichnung.
2. Das Fenster hat keine Lücken in der Sequenz und keine Gateway-Session ohne Datenbank-Session oder umgekehrt, was die Lückenerkennung aus Kapitel 14 ist. Eine Lücke innerhalb des Fensters ist eine Strecke, auf der die Antwort unbekannt ist.
3. Die Nebenpfade des Principals werden getrennt geprüft: Wiederherstellungen, Exporte und Kopien, die er angefordert hat (Kapitel 13), gefunden über das Register der Kopien und nicht über diese Abfrage.

Schlägt eine Prüfung fehl, ist das Ergebnis eine untere Grenze, und ich würde das in der Incident-Aufzeichnung in genau diesen Worten sagen. Die Abfrage hat außerdem eigene Grenzen. `rows_returned` ist eine Anzahl, keine Liste: Es sagt Ihnen, dass eine Restricted-Tabelle im Fenster eine bestimmte Zahl von Zeilen zurückgegeben hat, und nicht, welche Kunden das waren. Die Felder `fingerprint` und `literals_hash` erlauben die Frage, ob eine bekannte Anweisung wiederholt wurde, aber die Literale sind absichtlich geschwärzt. Die Aufzeichnung begrenzt also die Gefährdung und zählt sie nicht auf. Ein `rows_returned` ungleich null auf einer Restricted-Tabelle ist die Zeile, die die Sicherheitskontakte und die Rechts- und Compliance-Verantwortlichen der lesenden Person brauchen, und die Auslegung, was sie bedeutet, einschließlich jeder Benachrichtigungspflicht, liegt bei ihnen.

Der Lebenszyklus des Incident-Zugriffs

Die Eindämmung entfernt Zugriff. Ein Incident braucht auch hinzugefügten Zugriff: Jemand muss nachsehen, und jemand muss womöglich reparieren. Das Risiko in dieser Richtung ist der Grant, der in Eile angelegt wurde und am Montag noch da ist. Das Mittel ist ein Lebenszyklus mit sichtbarem Ende, in dem jedes Stück Zugriff, das für einen Incident geschaffen wurde, dieselben fünf Zustände durchläuft.

Das Diagramm ist eine Sicht auf die Aufzeichnungen, die Kapitel 8 bereits führt, kein neues Statusfeld. `granted` ist `approved` im Antragsdatensatz, `used` bedeutet, dass unter dem Antrag eine Anweisung aufgezeichnet wurde, und `reviewed` bedeutet, dass jemand anderes als die nutzende Person die Aufzeichnung gelesen hat. Achten Sie auf die beiden Ausgänge, die nicht bei `reviewed` ankommen: ein Grant, der über sein Ende hinaus bleibt, und ein Review, das nie stattfindet.



Damit das funktioniert, trägt jeder Antrag, der während eines Incidents gestellt wird, die Kennung des Incidents in seiner Begründung. Das Formular des Gateways kann sie verlangen, und ein Antrag ohne sie wird bei einem offenen Incident abgelehnt. Diese eine Konvention ermöglicht es einem Job zu fragen: „alles, was dieser Incident gewährt hat“.

```
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)
```

Dieser Job ersetzt weder die Prüfung auf überfällige Grants noch die Drift-Prüfung aus den Kapiteln 8 und 3. Er betrachtet dieselben Fakten von der Seite des Incidents aus und gibt dem Incident einen Kontrollpunkt: Niemand kann ihn als geschlossen markieren, solange ein Grant noch aktiv oder eine Break-Glass-Nutzung nicht geprüft ist. Ein Team, das ein Ticket-System nutzt, kann `closeIncident` mit dem Ticket verdrahten, was aus „wir haben vergessen zu widerrufen“ ein sichtbares Hindernis macht.

Reviewed

Der letzte Zustand ist der, der am häufigsten übersprungen wird, weil der Incident sich vorbei anfühlt. Das Review ist das, das Kapitel 8 für Break-Glass beschreibt, ausgedehnt auf jeden Grant, den der Incident erzeugt hat. Es ist schuldzuweisungsfrei und fragt nach dem System: War der Zugriff der kleinste, der die Aufgabe erledigen konnte, und gab es einen schnelleren Pfad, der hätte existieren sollen? Der Anhang enthält eine Vorlage für das Post-Incident-Review. Der Befund, der sich auszahlt, ist meist ein fehlender Pfad.

Schreibvorgänge während eines Incidents

Die Korrektur ist die verlockendste Stelle, eine Kontrolle zu lockern, weil der Schaden sichtbar ist und die Daten genau jetzt falsch sind. Kapitel 10 hat festgelegt, dass eine Korrektur eine Datenkorrektur ist, und Kapitel 8 hat festgelegt, dass Break-Glass kein Weg für Korrekturen ist. Unter Druck würde ich beide Entscheidungen beibehalten und den konformen Pfad schnell machen, statt den falschen Pfad verfügbar zu machen.

Die erste Entscheidung ist, zu welchem Pfad der Bedarf gehört. Die Prüfungen laufen von der engsten Tür zur weitesten.

```
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)"
```

Break-Glass ist nur für die kleine begrenzende Maßnahme erreichbar: ein Flag, das einen Job daran hindert, fehlerhafte Zeilen zu schreiben, ein Status, der einen Batch anhält. Die dauerhafte Korrektur der Daten ist immer eine Datenkorrektur, und zwar mit derselben Begründung wie zuvor: Eine Korrektur in einer rohen Break-Glass-Session überspringt die eine Kontrolle, die die Anweisung liest, bevor sie läuft, und läuft dort, wo ein unbegrenztes `UPDATE` noch möglich ist (Kapitel 8 nennt das ein akzeptiertes Risiko). Bei einem Incident ist die Versuchung stärker, und ebenso die Kosten eines einzigen falschen Schreibvorgangs.

Die Datenkorrektur behält ihre Form. Der Antrag benennt eine geprüfte Korrektur und trägt einen `approved_hash` aus dem Merge-Datensatz, der Runner lehnt ein Skript ab, dessen Hash abweicht, der Grant wird aus der Korrektur abgeleitet, und die Mengenschranke und der Snapshot gelten weiterhin (Kapitel 9 und 10). Was der Incident ändert, ist das Review.

```
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
```

Das spiegelt die beschleunigte Spur, die Kapitel 11 den Migrationen gibt: derselbe Job, dasselbe Credential, dieselbe Aufzeichnung, anderes Timing. Im Folge-Review wird die Abkürzung zurückgezahlt, und es macht aus einem beschleunigten Pfad auch ein gezähltes Ereignis. Stellt ein Team fest, dass die meisten Korrekturen eines Quartals ihn nutzen, ist die normale Spur für die Arbeit zu langsam, und die richtige Reaktion ist, die normale Spur zu beschleunigen, nicht aufzuhören zu zählen.

> **Warnung:** Eine Korrektur, die Break-Glass „braucht“, ist ein Signal für etwas anderes, wie Kapitel 10 sagt: eine Obergrenze, der eine Spalte fehlt, ein Genehmigerpool, der für die Uhrzeit zu klein ist, oder eine Restricted-Spalte, die auf den Produktpfad gehört. Bei einem Incident ist dieses Signal leicht zu ignorieren und teuer, es zweimal zu ignorieren.

Die Fälle, die frühere Kapitel zurückgestellt haben

Drei Situationen wurden von früheren Kapiteln weitergereicht. Die Tabelle ist die Entscheidung vorab; die Unterabschnitte geben die Begründung.

Situation	Was stoppt	Was sich nicht ändert	Vorab entschieden
Das Gateway ist ausgefallen	Menschlicher Datenzugriff (keine zweite Tür)	Services laufen weiter mit ihren eigenen Credentials	Wer das Gateway wiederherstellt, und wann das versiegelte Instrument genutzt werden darf
Während des Incidents ist eine Wiederherstellung nötig	Nichts, wenn der Pfad geübt wurde	Genehmiger, isoliertes Ziel, Registereintrag, geplante Löschung	Ein benannter Genehmiger, ein vorbereitetes Ziel, ein Runbook zur Wiederherstellung
Genehmigungen sind nachts langsam	Eine Korrektur, die kein Notfall ist	Selbstgenehmigung wird weiterhin abgelehnt	Eine Eskalationsreihenfolge und wer den Pool erweitern darf

Das Gateway ist ausgefallen

Die Kapitel 7 und 8 haben entschieden, dass es keine zweite Tür gibt: Menschen erreichen die Datenbank nur über das Gateway, also ist ein Ausfall des Gateways auch ein Ausfall von Break-Glass. Die Begründung dort war, dass ein Credential, das ohne das Gateway funktioniert, auch ohne das Logging, die Identität und den Ablauf funktioniert. Daran würde ich in der schlimmsten Nacht festhalten, und der erste Schritt bei einem Ausfall ist, zu bemerken, was nicht gestoppt hat. Die Services verbinden sich mit ihren eigenen Credentials und laufen weiter, daher brauchen viele Incidents, bei denen das Gateway zufällig ausgefallen ist, überhaupt keine Person in der Datenbank. Das Deployment zurückrollen, den Job pausieren, das Feature-Flag umlegen, den Service skalieren: Nichts davon läuft über das Gateway.

Braucht der Incident tatsächlich Datenzugriff, sind es jetzt zwei Incidents, und der erste ist das Gateway. Seine Wiederherstellung kommt vor der Datenfrage, und die Menschen, die es wiederherstellen, sind das Platform-Team, das den Deploy-Pfad und die Redundanz nutzt, die Kapitel 7 verlangt hat.

Sind alle Instanzen ausgefallen, existiert kein menschlicher Datenpfad, und das eine verbleibende Instrument ist der Superuser, den Kapitel 3 als versiegelt behandelt hat. Er ist ein Werkzeug auf Engine-Ebene und kein Datenpfad: Er umgeht die Append-only- und Restricted-Schutzmechanismen, und seine Netzwerkroute ist für Menschen geschlossen. Ich würde

seine Nutzung darauf beschränken, das Gateway oder das, wovon das Gateway abhängt, wiederherzustellen, nie darauf, Geschäftsdaten zu lesen oder zu schreiben, und ich würde die Regel aufschreiben.

> **Warnung:** Nutzen Sie das versiegelte Instrument nicht, um Daten anzusehen, weil das Gateway ausgefallen ist. Es umgeht die Append-only- und Restricted-Schutzmechanismen, und der Zugriff, den es gibt, ist der Zugriff, den alle früheren Kapitel verhindern sollten. Wenn Daten gelesen werden müssen, stellen Sie zuerst das Gateway wieder her.

Seine Nutzung unter dieser Regel hat einen Mechanismus. Der Abruf des Credentials hinterlässt eine Aufzeichnung, eine zweite Person ist anwesend und benannt, das Öffnen der Netzwerkroute ist eine Änderung mit einem Verantwortlichen und einer Endzeit, und das Credential wird danach rotiert, weil eine Person es gesehen hat. Das Gateway zeichnet in diesem Zeitfenster nicht auf, daher ist das Engine-Log die einzige Spur. Nachdem das Gateway wieder da ist, hängt das Platform-Team von Hand einen Eintrag an, wie den externen Audit-Eintrag, den Kapitel 14 für Migrationen beschreibt, und die Nutzung bekommt das Review, das jede Break-Glass-Nutzung bekommt.

Eine Wiederherstellung während eines Incidents

Kapitel 13 hat die Wiederherstellung zu einem Antrag gemacht: ein Genehmiger, der nicht der Antragsteller ist, ein isoliertes Ziel, ein Registereintrag, eine geplante Löschung und ein geschlossener Start. Die dort genannten Kosten waren die Geschwindigkeit bei einem Incident, und die Antwort war, zu üben und die Genehmigung vorab zu regeln. Das ist der gesamte Mechanismus, daher lohnt es sich auszuführen, was „vorab“ bedeutet.

Vor dem Incident benennt das Runbook, wer nachts eine Wiederherstellung genehmigen kann, das Ziel ist vorbereitet und isoliert, und der Wiederherstellungsantrag hat ein Formular, das eine Incident-Kennung aufnimmt. Während des Incidents läuft der Antrag unverändert durch: Der Genehmiger ist nicht der Antragsteller, der Registereintrag kommt zuerst,

die geplante Löschung ist gesetzt, bevor die Kopie existiert, und Logins sind deaktiviert und die Zugriffsdefinition ist angewendet, bevor die Kopie geöffnet wird. Der Incident fügt die Begründung hinzu, keine Abkürzung.

Wofür die Kopie da ist, zählt mehr als wie sie erstellt wird. Bei einem Incident ist eine wiederhergestellte Kopie eine Lesequelle, meist für alte Werte: den Zustand einer Spalte vor einem falschen Schreibvorgang oder versehentlich entfernte Zeilen. Ihre Leser sind die Personen auf dem Antrag. Die Korrektur, die folgt, läuft gegen den Primary als Datenkorrektur, wobei die aus der Kopie gelesenen Werte als Eingabe für ein geprüftes Skript dienen, und `reapplyDeletions` aus Kapitel 13 läuft, bevor die Kopie geöffnet wird, damit die Kopie keine Daten zurückbringt, die absichtlich gelöscht wurden. Den Primary durch eine wiederhergestellte Kopie zu ersetzen, ist eine Entscheidung zur Wiederherstellung und gehört dem Platform-Team und dem Incident Commander. Die darin enthaltene Zugriffsfrage ist die oben genannte: wer die Kopie liest und wann sie gelöscht wird.

Langsame Genehmigungen in der Nacht

Das Nachtszenario aus Kapitel 8 endete in einer Entscheidung, die dieses Kapitel beibehält: Wenn der Genehmigerpool nicht antwortet und der Bedarf warten kann, hinterlassen Sie eine Notiz und hören auf. Das ist ein gutes Ergebnis und eines, das das alte Ledgerly nicht hervorbringen konnte. Für einen Bedarf, der nicht warten kann, ist die Eskalation in den Antrag eingebaut, sodass ein Page, der unbeantwortet bleibt, innerhalb desselben Antrags im Pool weiterwandert.

```
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
```

Zwei Regeln umgeben das. Erstens wird der Pool nur von einer Person erweitert, der er gehört, und die Erweiterung ist eine Aufzeichnung: Bei einem erklärten Incident darf der Incident Commander für den Incident einen benannten Genehmiger hinzufügen, mit einem Ende. Zweitens gibt es weiterhin keine automatische Genehmigung für die diensthabende Person, denn eine Kontrolle, die jeden Antrag innerhalb von Sekunden gewährt, ist keine Kontrolle. Wenn Genehmigungen regelmäßig das Grant-Fenster überdauern, betrifft der Befund den Dienstplan, die Größe des Pools oder die Uhrzeit, und das Mittel ist, Personen zum Pool hinzuzufügen, nicht, den Genehmiger aus der Schleife zu nehmen.

Die Pfade üben

Ein Pfad, der nie gelaufen ist, ist ungetestet. Das ganze Kapitel besteht aus Schritten, die niemand zum ersten Mal während eines Incidents entdecken möchte, also braucht jeder eine Übung, hier Game Day genannt: eine geplante Übung mit einem Szenario, einem erwarteten Ergebnis und einer Prüfung der Aufzeichnung. Vier Szenarien decken ab, was dieses Kapitel gebaut hat. Sie sind bewusst defensiv, jedes beginnt mit einem Auslöser, den das Monitoring bereits zu erkennen weiß, und jedes trägt eine beobachtbare Bestehensbedingung.

Szenario	Auslöser	Erwartetes Verhalten	Bestehensbedingung	Geübte Teile
Credential-Leak	Das Credential eines Übungs-Principals wird als offengelegt erklärt	`containCompromise` in der Reihenfolge; Abfrage des Umfangs am Incident angehängt	Kein aktiver Login für den Principal, unfinished ist leer, der Umfang benennt jede Tabelle, die die Übung berührt hat	Identität, Grant, Aufzeichnung

Szenario	Auslöser	Erwartetes Verhalten	Bestehensbedingung	Geübte Teile
Gateway ausgefallen	Erst eine, dann alle Gateway-Instanzen in einer Nicht-Produktionskopie gestoppt	Menschlicher Zugriff stoppt; Services laufen weiter; der Wiederherstellungspfad läuft	Das Plattform-Team stellt das Gateway mithilfe des Runbooks wieder her; versiegeltes Instrument ungenutzt oder unter seiner Regel mit Aufzeichnung genutzt	Pfad, Aufzeichnung
Verdächtiger Export	Ein Alarm bei großem `rows_returned`, aus `evaluateAlertRules` (Kapitel 14)	Den Principal eindämmen; das Ziel im Register der Kopien finden; löschen	Die Kopie ist gefunden, beendet und aufgezeichnet; die Abfrage des Umfangs stimmt mit dem Register überein	Identität, Aufzeichnung, Ausführung
Fehlerhaftes `UPDATE` über viele Zeilen	Ein Schreibvorgang, der in die Row Cap passt und die falschen Zeilen ändert, in einer Übungsumgebung	Weitere Schreibvorgänge stoppen; die Umkehrung aus dem Snapshot berechnen; als beschleunigte Datenkorrektur ausführen	Umkehrung von einer Reviewerin oder einem Reviewer per Merge übernommen, die oder der nicht der Autor ist; Folge-Review eingeplant	Ausführung, Grant, Aufzeichnung

Ein Game Day gilt nur nach dem beobachteten Ergebnis als bestanden, nie, weil sich jemand daran erinnert, was zu tun ist. Wenn eine Übung endet, weil jemand eine Abkürzung kannte, ist sie durchgefallen, und die Ab-

kürzung wird zu dem, was zu bauen ist. Schreiben Sie die Zeiten auf, die jeder Schritt tatsächlich gebraucht hat; diese Zahlen gehören Ihrem Unternehmen, und ich würde keine Zahl nennen, die es nicht gibt.

Nutzen Sie nach Möglichkeit eine Nicht-Produktionskopie und synthetische Principals bei Übungen in der Produktion, damit keine echten Kundendaten gelesen werden, um das Lesen zu üben. Übungskopien werden registriert und geschlossen wie jede andere Kopie (Kapitel 13).

Abwägungen und Kosten

Die erste Kostenart ist die Übung selbst. Game Days holen Entwicklerinnen und Entwickler von der Produktarbeit weg, und eine Übung braucht eine verantwortliche Person und einen Kalender. Für ein Team von drei ist eine einzelne Übung pro Jahr, die den Credential-Leak abdeckt, ein vernünftiger Anfang, und die anderen drei Szenarien können warten, bis das Team wächst.

Die zweite Kostenart ist der Ausfall, den die Eindämmung verursacht. Das sofortige Widerrufen des Credentials eines Services ist ein Ausfall in diesem Service, und die Eindämmung von `ledgerly_access_admin` stoppt die Bereitstellung. Beides sind Entscheidungen, die vorab mit den Menschen zu treffen sind, die sie tragen, denn die Person, die um 3 Uhr nachts entscheidet, sollte nicht auch erst lernen müssen, was die Entscheidung kostet.

Die dritte Kostenart ist die Konvention der Incident-Kennung. Sie fügt jedem Antrag ein Feld und einen weiteren Grund für die Ablehnung eines Formulars hinzu, und sie ist nur so gut wie die Bereitschaft der Menschen, sie auszufüllen. Der Abgleich, der von ihr abhängt, scheitert leise, wenn sie nicht ausgefüllt wird, daher sollte das Gateway das Feld erzwingen und sich nicht auf Gewohnheit verlassen.

Die vierte Kostenart ist Ehrlichkeit beim Umfang: Manche Incidents enden mit einer Grenze, die größer ist als die Wahrheit, und ich würde die Grenze berichten, statt eine Genauigkeit zu behaupten, die der Aufzeichnung fehlt.

Wenden Sie nicht die volle Maschinerie auf ein System an, das noch kein Audit-Log hat. `blastRadius` braucht die Aufzeichnung, und `reconcileIncidentAccess` braucht die Antragsdatensätze; ein Team ohne sie sollte zuerst diese bauen (Kapitel 8 und 14) und in der Zwischenzeit ein kurzes Runbook schreiben, das benennt, wer widerruft, wer Sessions schließt und wer die Frage verantwortet, was berührt wurde.

Häufige Fehler

- **Rotieren, ohne zu widerrufen.** Eine Rotation ändert das Secret und lässt die offene Session und die gewährten Privilegien bestehen. Sie ist verlockend, weil die Rotation eine Routine ist, die das Team schon hat. Widerrufen Sie zuerst, schließen Sie dann die Sessions, rotieren Sie danach.
- **Eingrenzen, bevor eingedämmt wird.** Der Drang zu verstehen kommt vor dem Drang zu stoppen, und ein Team kann eine Stunde mit der Abfrage verbringen, während der Zugriff aktiv bleibt. Dämmen Sie zuerst ein. Die Aufzeichnung überlebt die Eindämmung, weil sie in einem Speicher liegt, den der Zugriff nicht berühren kann.
- **Dem Log vertrauen, ohne es zu prüfen.** Eine Abfrage des Umfangs über ein Fenster mit einer Lücke liest sich wie eine saubere Antwort. Prüfen Sie zuerst die Kette und die Sequenz, und berichten Sie eine untere Grenze, wenn eines von beiden fehlschlägt.
- **Break-Glass als Spur für Korrekturen.** Es ist schnell und verfügbar, und es hat kein Review der Anweisung. Nutzen Sie es für die kleine begrenzende Maßnahme und führen Sie die Korrektur als beschleunigte Datenkorrektur aus.
- **Incident-Zugriff offen lassen.** Der Incident fühlt sich vorbei an, wenn die Ursache behoben ist, und die Grants bleiben. Binden Sie das Schließen des Incidents an den Abgleich, damit das Ticket nicht geschlossen werden kann, solange Zugriff offen ist.

Zentrale Erkenntnisse

- Dringlichkeit verändert die Geschwindigkeit einer Kontrolle, nie die Identität, die handelt, die Anweisung, die läuft, oder die Aufzeichnung, die bleibt.

- Die Eindämmung hat eine Reihenfolge: Grants widerrufen, Sessions schließen, Credentials rotieren, anhand des Audit-Logs eingrenzen, dann aufzeichnen und benachrichtigen, und jeder Schritt läuft, auch wenn der vorherige fehlgeschlagen ist.
- Widerruf ist nicht Rotation: Eine Rotation stoppt neue Logins und tut für die bereits offenen nichts, weshalb ein geleaktes statisches Credential zuerst getrennt und danach ersetzt wird.
- Der Auswirkungsbereich stammt aus einer einzigen Abfrage über `access_audit_record`, und er ist eine untere Grenze, sobald die Kette oder die Sequenz eine Lücke hat.
- Incident-Zugriff durchläuft requested, granted, used, revoked und reviewed, und ein Incident kann nicht geschlossen werden, solange ein von ihm erzeugter Grant noch offen ist.
- Korrekturen bei einem Incident sind Datenkorrekturen mit beschleunigtem Review, und Break-Glass bleibt der kleinen Maßnahme vorbehalten, die Schaden stoppt.
- Ein Pfad, der nicht geübt wurde, ist eine Annahme; daher bekommen der Credential-Leak, der Ausfall des Gateways, der verdächtige Export und das fehlerhafte `UPDATE` über viele Zeilen je einen Game Day mit beobachtbarer Bestehensbedingung.

Kapitel 16 fasst die Kontrollen des ganzen Buches in einer Checkliste und einer Reifegradleiter zusammen und kehrt zu dem Satz zurück, dem jedes Kapitel gedient hat.

Kapitel 16 — Eine Checkliste für den Zugriff und Schlussgedanken

Stellen Sie sich Ledgerly ein Jahr nach dem Sicherheitsfragebogen aus Kapitel 1 vor. Ein weiterer potenzieller Kunde schickt dieselben zwei Zeilen: „Listen Sie alle Personen und Systeme auf, die sich mit Ihrer Produktionsdatenbank verbinden können“ und „Woher wissen Sie, was sie getan haben?“ Alex öffnet statt des Passwortmanagers das Zugriffsinventar, liest die Liste der Türen und schickt sie an einem Nachmittag zurück. Nichts an der Antwort beeindruckt. Sie ist kurz, jede Zeile hat einen Verantwortlichen und ein Ende, und nichts davon erforderte Gräberei.

Robin, in dieser Woche die diensthabende Person, fragt per Chat: „Wir haben ein Gateway, einen Runner, eine Pipeline, einen Audit-Speicher und drei Review-Jobs. Welche davon haben wir gebraucht, und welche haben wir gebaut, weil das Buch es verlangt hat?“ Alex hat keine saubere Antwort. Manche Teile haben sich ihren Platz verdient, als zum ersten Mal ein Grant von selbst ablief; andere wurden nie im Ernstfall gebraucht, und Alex kann nicht sagen, ob das heißt, dass sie funktionieren oder ungenutzt herumstehen.

Dieses Kapitel dient allen drei Säulen zugleich: Autorisierung, Ausführung und Zurechenbarkeit. Es fügt keine neue Kontrolle hinzu. Es fügt die Stufen des Zielzustands hinzu, in der Reihenfolge, in der ein Team sie erreichen kann, die Patterns nebeneinander bewertet und eine Checkliste, deren Zeilen Beobachtungen statt Meinungen sind. Die Frage, die es beantwortet, ist die von Robin: Wie entscheidet ein Team, was es als Nächstes baut, und woher weiß es, dass eine Kontrolle funktioniert?

Das Problem und warum es schwierig ist

Fünfzehn Kapitel hinterlassen bei Lesenden viele Kontrollen und keine Reihenfolge. Jede wurde mit eigener Begründung eingeführt, und jede Begründung war in ihrem Kapitel stichhaltig. Zusammen gelesen wirken sie

wie ein Projekt von einem Jahr oder mehr, und ein Team, das vor dieser Liste steht, tut tendenziell eines von zwei Dingen. Es beginnt mit dem Billigsten und hört auf, sobald die Arbeit politisch wird, oder es beginnt mit dem Beängstigendsten und baut ein aufwendiges Stück des Entwurfs, während das gemeinsam genutzte Passwort noch in fünf Konfigurationsdateien steht.

Die Schwierigkeit liegt darin, dass „fertig“ in der Zugriffsarbeit schwer zu erkennen ist. Ein Rollenschema, das auf dem Papier existiert, und eines, das durchgesetzt wird, sehen in einem Review-Meeting gleich aus. Deshalb messen Teams daran, was installiert wurde, und installiert ist die falsche Einheit. Die nützliche Einheit ist ein Test, den jemand ausführen kann: eine Anweisung, die sich zurückverfolgen lässt, eine Verbindung, die abgelehnt wird, ein Grant, der verschwunden ist.

Es ist auch eine Frage dessen, wie das System gebaut ist, und nicht dessen, was die Richtlinie sagt. Eine Richtlinie legt fest, was geschehen soll; ein Entwurf bestimmt, was geschehen kann und was eine Spur hinterlässt, wenn es geschieht. Jede der folgenden Zeilen lässt sich beantworten, indem man ein System, ein Log oder eine Tabelle ansieht, und eine Zeile, die sich nur beantworten lässt, indem man jemanden nach seinen Absichten fragt, steht nicht auf der Liste.

Die These, konkretisiert

Das Buch hat einen einzigen Satz vertreten, und hier wird er an den Kapiteln überprüft:

> Der Zugriff auf Produktionsdatenbanken ist kein Problem der Credential-Verteilung; es ist ein Problem der Autorisierung, der Ausführung und der Zurechenbarkeit.

Der Satz hat drei Säulen, und jede Säule steht auf bestimmten Kapiteln. Autorisierung ist Grant plus Pfad, pro Anfrage entschieden. Ausführung ist das, was mit jeder Anweisung geschieht. Zurechenbarkeit ist Identität plus Aufzeichnung. Ein Passwort berührt nur die erste davon, und nur an der Tür.

Säule	Teile des Modells	Kapitel, die sie aufbauen	Die Frage, die Sie nun beantworten können
Autorisierung	Grant, Pfad	3, 5, 6, 8, 12	Was darf dieser Principal tun, wo, wie lange und mit welchen ausgeblendeten Spalten?
Ausführung	Ausführung	7, 9, 10, 11, 13	Wie gelangte diese Anweisung an die Daten, wer oder was hat sie geprüft, und was hat sie begrenzt?
Zurechenbarkeit	Identität, Aufzeichnung	4, 7, 14, 15	Welche Person oder welche Workload-Instanz hat gehandelt, auf wessen Antrag, und welcher Nachweis verbindet sie mit dem Ergebnis?

Die Kapitel 1 und 2 rahmen alle drei, und Kapitel 15 prüft sie unter Druck. Kapitel 7 erscheint zweimal, weil das Access-Gateway sowohl der Pfad ist als auch der Ort, an dem aus einer Person ein Session-Login wird, den ein Log benennen kann; Kapitel 13 steht unter der Ausführung, weil der Export-Schritt von `reporting-job` eine Anweisungsmenge ausführt und die anderen Seitentüren Umgehungen derselben Kontrollen sind.

Warum die Arbeit an den Credentials notwendig war und doch nicht die Antwort

Kapitel 4 handelt von Credentials, und es ist das Kapitel, das die meisten Lesenden als Mittelpunkt des Buches erwarten. Es ist notwendig. Ein Service, der sein Passwort aus einer gemeinsam genutzten Datei liest, hat keine eigene Identität, und ohne eigene Identitäten kann die Aufzeichnung niemanden benennen. Kurzlebige und dynamische Credentials machen den Widerruf zu einem Routineschritt. Workload Identity beseitigt die Datei.

Nichts davon entscheidet, was ein Principal tun darf, sobald er drin ist, wie eine Anweisung geprüft wird oder was festgehalten wird. Ein Team könnte Kapitel 4 perfekt abschließen und trotzdem eine Rolle haben, der jede Tabelle gehört, eine Laptop-Session, die jede Anweisung so mit einem Commit abschließt, wie sie getippt wurde, und ein Log, in dem jede Zeile einen eigenen Namen trägt und keine einen Grund. Ein Passwortspeicher liefert die eigenen Namen, die der Teil Identität braucht, und über den Widerruf ein Stück Grant; ein Passwort berührt nur die erste Säule, und nur an der Tür. Die anderen Teile werden anderswo gebaut, und der Rest des Buches zeigt, wo.

Wenn ein Vorschlag lautet „wir beheben den Zugriff, indem wir X verbessern“, ordnen Sie X im Modell ein, bevor Sie zustimmen:

```
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
```

Die Funktion ist kein Gate. Sie benennt, welche Teile ein Vorschlag unverändert lässt, denn ein leerer Teil ist ein Befund und keine Auslassung.

Ein Katalog von Patterns, bewertet nach Säulen

Kapitel 1 zeichnete die Bandbreite der Wege hinein, und Kapitel 7 verglich die konkreten Pfade. Die folgende Tabelle stellt die fünf Patterns dieser Bandbreite nebeneinander und bewertet jedes gegen die drei Säulen. Die Bewertungen sind mein Urteil und keine Messung. Sie sagen, wie viel von einer Säule das Pattern von sich aus belegen kann, wenn die Grants und das Logging der früheren Kapitel gleich gehalten werden. Lesen Sie „stark“ als „das Pattern ist dafür gebaut, dies zu beweisen“, „teilweise“ als „es beweist einen Teil davon, oder nur, wenn etwas hinzukommt“ und „keine“ als „das Pattern schweigt hier“.

Pattern	Autorisierung	Ausführung	Zurechenbarkeit
Direktzugriff (Client zur Datenbank)	Teilweise: nur Grants, die Reichweite ist, was das Netzwerk erlaubt	Keine	Bestenfalls teilweise: Logins pro Person, wenn jemand sie bereitstellt, sonst keine
Bastion	Teilweise: verengt den Netzwerkpfad	Keine: Sie leitet weiter und kann ein `SELECT` nicht von einem `DROP` unterscheiden	Teilweise: wer sich am Host angemeldet hat, nicht was lief

Pattern	Autorisierung	Ausführung	Zurechenbarkeit
Session-Proxy oder Access-Gateway	Stark: Session-Logins pro Person, Ablauf, Grants pro Antrag	Teilweise: begrenzt durch Grants und Guarded Settings, Anweisungen werden frei getippt	Stark bei wer und wann, teilweise bei was
Anweisungsvermittelnder Executor	Stark	Stark: vor der Ausführung geprüft, an einen Approved Hash gebunden	Stark: Anweisung, Genehmiger und Ergebnis in einer Aufzeichnung
Produktpfad (kein SQL)	Stark: die Operation ist die Berechtigung	Stark: die Operation ist festgelegt	Teilweise: die Person steht im Prüfpfad des Tools, das Datenbank-Log zeigt den Service-Login

Direktzugriff

- **Beweist:** dass ein Login existiert und dass seine Grants so waren, wie sie waren.
- **Kann nicht beweisen:** welche Person einen gemeinsam genutzten Login verwendet hat oder was eine Anweisung bewirkt hat.
- **Kosten:** keine im Aufbau, und das ist der Reiz. Jede Kostenstelle kommt später, als eine Woche Gräberei.
- **Ausfallmodus:** das gemeinsame Admin-Passwort aus Kapitel 1.
- **Nicht einsetzen, wenn:** mehr als eine Person oder Workload den Login erreichen kann. Für ein System mit einem einzigen Entwickler, dessen einziger Consumer sein Verantwortlicher ist, ist es tragbar, und ich würde aufschreiben, was dazu führen würde, dass das nicht mehr gilt.

Bastion

- **Beweist:** dass eine Verbindung über einen kontrollierten Host kam und, wenn die Konten darauf pro Person geführt werden, wer sich dort angemeldet hat.
- **Kann nicht beweisen:** was eine Anweisung war, da der Host das Datenbankprotokoll nicht spricht.
- **Kosten:** ein Host, der mit Patches versorgt und verfügbar gehalten werden muss, im Tausch gegen eine einzige Stelle, an der sich der Netzwerkpfad abschalten lässt.

- **Ausfallmodus:** Ein gemeinsames Bastion-Konto macht den Host wieder zum gemeinsamen Passwort, nur einen Hop früher.
- **Nicht einsetzen, wenn:** Sie Nachweise auf Anweisungsebene brauchen. Als Netzwerkreichweite vor etwas, das aufzeichnet, bleibt sie nützlich.

Session-Proxy oder Access-Gateway

- **Beweist:** dass eine benannte Person zwischen zwei Zeitstempeln unter bestimmten Grants einen bestimmten Session-Login innehatte.
- **Kann nicht beweisen:** dass eine in dieser Session getippte Anweisung eine gute Idee war. Die Session ist ein Handout (Kapitel 9), und alles, was die Grants erlauben, kann getippt werden.
- **Kosten:** ein neues System, das verfügbar sein muss, wenn jemand es braucht, und der Genehmigungspfad, der dazugehört (Kapitel 7 und 8).
- **Ausfallmodus:** ein Gateway mit einer zweiten Tür daneben, also ein gemeinsames Passwort mit besserem Namen.
- **Nicht einsetzen, wenn:** die Aufgaben sich wiederholen und eine feste Form haben. Sie gehören auf einen Produktpfad, wo niemand eine Session braucht.

Anweisungsvermittelnder Executor

- **Beweist:** dass genau diese Anweisungsmenge, identifiziert durch ihren Hash, von einer benannten Person genehmigt, geprüft, innerhalb einer Schranke ausgeführt wurde und ein gezähltes Ergebnis lieferte.
- **Kann nicht beweisen:** dass die Analyse die Anweisung verstanden hat. Die Abfrageanalyse verringert Fehler und ist nicht die Autorisierungsgrenze; diese bleibt bei den Grants (Kapitel 9).
- **Kosten:** Jede Aufgabe wartet auf ein Review, nichts ist interaktiv, und jemand muss den Runner bauen und pflegen.
- **Ausfallmodus:** eine Genehmigung, die innerhalb von Sekunden immer erteilt wird, also Sicherheitstheater mit zusätzlichen Schritten.
- **Nicht einsetzen, wenn:** die Arbeit explorativ ist. Wer einen Zustand verstehen will, braucht zuerst eine Lese-Session, und dafür ist `human_readonly` da.

Produktpfad

- **Beweist:** dass eine festgelegte Operation lief, für eine Rolle in einem Tool, mit dem eigenen Prüfpfad des Tools.
- **Kann nicht beweisen:** irgendetwas über die Aufgaben, für die niemand ein Feature gebaut hat. Freie Fragen haben keinen Pfad.
- **Kosten:** ein Feature pro Aufgabe und der langsamste Änderungszyklus der fünf.
- **Ausfallmodus:** der Produktpfad, der eine „Beliebiges SQL ausführen"-Notluke ansammelt, weil der erste dringende Fall kein Feature hatte.
- **Nicht einsetzen, wenn:** die Aufgabe sich noch nicht wiederholt hat. Kapitel 10 beschreibt das Signal dafür, eine wiederholte Korrektur zu einem Feature zu machen, und eine Aufgabe, die einmal vorkam, ist dieses Signal nicht.

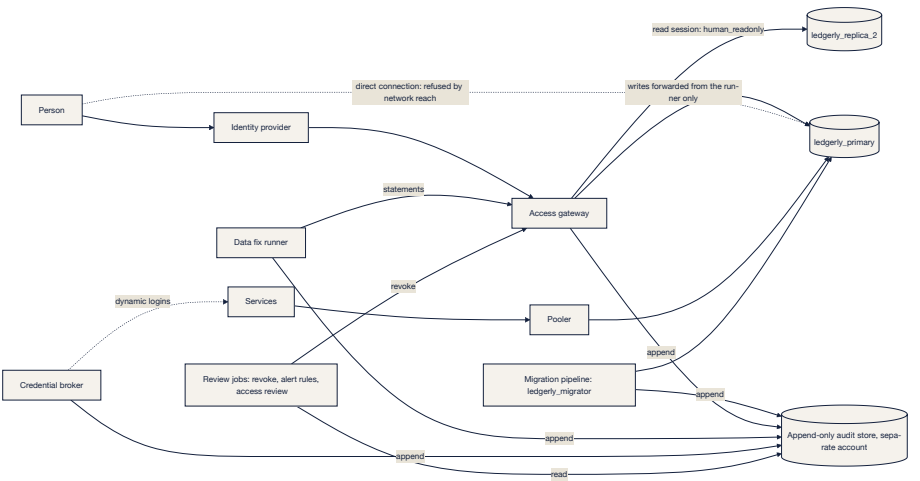
Das Pattern ganz rechts ist nicht immer das beste, und eine Frage zu den Rechnungen eines einzelnen Kunden braucht keinen anweisungsvermittelnden Executor. Ich würde pro Aufgabe wählen, mit dem günstigsten Pattern, das die Säulen, die die Aufgabe gefährdet, noch belegt:

```
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"
```

Die letzte Zeile ist der Kern. Ein Weg, der nicht passt, kommt mit einem Verantwortlichen und einem Enddatum ins Inventar, wie der Inventarblock aus Kapitel 1 es verlangt, und eine Ausnahme ohne beides ist ein Überbleibsel.

Die Referenzarchitektur

Das Diagramm zeigt den Zielzustand von Ledgerly mit allen Kontrollen des Buches in einem Bild. Achten Sie auf drei Dinge: dass kein Pfeil von einem Menschen eine Datenbank erreicht außer über das Gateway, dass jeder Schreibvorgang einer Person den Runner passiert und dass der Audit-Speicher in einem Konto liegt, in das die Datenbankadministratoren nicht schreiben können.



Das Bild vereinfacht: Die Lese-Replikate der Anwendung aus Kapitel 6 teilen sich ein Kästchen mit ihren Writern, und der maskierte Analytics-Ladevorgang, der Export für den Anbieter und die Backups (Kapitel 13) sind weggelassen, weil jedes davon eine Kopie mit einem Verantwortlichen und einem Ablauf ist und eine Zeichnung mit allen Kopien keine lesbare Mitte hätte.

Jedes Kästchen lässt sich dem Modell zuordnen: Der Identity-Provider und der Broker liefern die Identität; Rollen, Session-Logins und der Ablauf-Job liefern den Grant; die Netzwerkregel hinter dem gepunkteten Pfeil und das Gateway liefern den Pfad; der Runner und die Pipeline liefern die Ausführung; der Audit-Speicher und die Review-Jobs liefern die Aufzeichnung. Ein Kästchen, das keinen Teil liefert, ist ein Kandidat für die Entfernung, und Robins Frage verdient diesen Test.

Eine Reifegradleiter mit Ausstiegstests

Der Zielzustand ist eine Richtung und keine Ziellinie, deshalb sind die Schritte des Buches hier in fünf Stufen gruppiert. Die Gruppierung stammt von mir. Sie folgt dem, wie viel ein Team ändern kann, bevor die nächste Änderung leichter wird, und andere Organisationen werden sie umordnen; die eine Reihenfolge, die ich verteidigen würde, ist, dass das Inventar zuerst kommt, denn ohne es weiß niemand, was jeder spätere Schritt abdecken muss. Auch die Aufzeichnung beginnt man am besten früh in einfacher Form, da sich jede spätere Stufe mit Nachweisen leichter beurteilen lässt.

Eine Stufe ist abgeschlossen, wenn ihr Ausstiegstest besteht, und „wir haben es installiert“ ist kein Test. Jeder Test ist eine Beobachtung, die sich wiederholen lässt.

Stufe	Was sie hinzufügt	Ausstiegstest	Warum Teams hier steckenbleiben
1. Die Türen kennen	Inventar von Principals, Türen, Kopien (Kapitel 1, 2, 13)	Ein Dokument listet jede Tür mit Verantwortlichem und Ende auf, und ein neuer Entwickler kann „wer erreicht den Primary?“ allein daraus beantworten	Das Inventar wird als Projekt behandelt, deshalb wird es nie fertig und nie für vertrauenswürdig gehalten
2. Jeden Principal benennen	Eigener Login und eigene Rolle pro Principal, dynamische Credentials, Budgets pro Rolle (Kapitel 3, 4, 5)	Ein Tag des Datenbank-Logs enthält keinen gemeinsam genutzten Login, und kein Service verbindet sich als Eigentümer oder als Admin	Die Umschaltung wirkt für jeden Service wie ein Ausfallrisiko, deshalb bleibt das gemeinsame Passwort „vorerst“
3. Ein Pfad, begrenzt	Gateway, Replikate, JIT und Break-Glass, Klassifizierung und Maskierung (Kapitel 6, 7, 8, 12)	Eine direkte Verbindung zum Primary von einem Laptop aus schlägt fehl, ein JIT-Grant verschwindet bei `expires_at`, ohne dass jemand handelt, und eine Lese-Session liefert keine Restricted-Spalte	Angst vor dem Gateway als Single Point of Failure und Genehmiger, die nachts nicht antworten können

Stufe	Was sie hinzufügt	Ausstiegstest	Warum Teams hier steckenbleiben
4. Vermittelt und festgehalten	Ausführungsrichtlinie, Runner, geprüfte Korrekturen, Migrationen nur über die Pipeline (Kapitel 9, 10, 11)	Ein Schreibvorgang vom eigenen Client einer Person wird abgelehnt, und der Runner lehnt eine Anweisungsmenge ab, deren Hash von `approved_hash` abweicht	Das Review wird zum Engpass, deshalb häufen sich Ausnahmen schneller als Produkt-Features
5. Nachweisbar und eingeübt	Audit-Speicher mit Hash-Kette, Alarmregeln, Zugriffs-Review, Game Days (Kapitel 14, 15)	Eine unangekündigt von gestern herausgegriffene Anweisung lässt sich einer Person, einem Antrag und einem Ergebnis zuordnen, ohne jemanden zu fragen; `verifyChain` besteht; ein Game Day endet mit einem aus der Aufzeichnung bestätigten Widerruf des Zugriffs	Das Log existiert und niemand liest es, und der Game Day ist das Erste, was gestrichen wird, wenn ein Release sich verspätet

Eine Stufe kann ihren Test nach der Abnahme wieder nicht bestehen, weil Konfigurationen driften; ein nach Zeitplan wiederholter Test zeigt die Drift, solange sie klein ist.

> **Hinweis:** Ein kleines Team mit einer Datenbank und vier Entwicklern könnte vernünftigerweise bei Stufe drei aufhören und aufschreiben, was dazu führen würde, dass es weitermacht. Entscheidend ist der schriftliche Grund, nicht die Höhe.

Die Checkliste

Die Checkliste ist nach Säulen gruppiert, und jede Zeile ist eine Beobachtung. Für die Verwendung gilt dieselbe Konvention wie für die Ausstiegstests: Ein „nein“ ist ein Befund mit Verantwortlichem und Datum, kein Versagen, das sich wegerklären ließe.

```
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
```

Autorisierung

- [ ] Jede Tabelle gehört `ledgerly_owner`, und keine Person und kein Service hält dessen Login.
- [ ] Jeder Service verbindet sich mit einem Login, der Mitglied seiner eigenen Rolle `app_*` ist, und keine Anwendungsrolle verfügt über DDL.
- [ ] Für keine Person existiert Dauerzugriff mit Schreibrechten; `human_readonly` ist der Standard, und `human_jit_write` ist eine Obergrenze und keine vergebene Rolle.
- [ ] Jeder JIT- und Break-Glass-Grant hat ein `expires_at`, und der Widerrufs-Job und seine unabhängige Prüfung auf Überfälliges liefen beide kürzlich.
- [ ] Eine direkte Verbindung zum Primary von einem Laptop aus wird abgelehnt, und die Replikate weisen den Admin-Login zurück.
- [ ] Restricted-Spalten tauchen in keinem JIT-Antrag, in keiner Lese-Session und in keiner Analytics-Kopie auf, und der Klassifizierungskatalog listet jede Spalte auf, die existiert.
- [ ] Jedes Backup, jeder Dump, jeder Export und jede Staging-Kopie hat einen benannten Verantwortlichen und einen Ablauf.

Ausführung

- [ ] Anweisungen von Personen und von Automatisierung erreichen die Daten über die Ausführungsrichtlinie, und die deny / gate / allow-Matrix liegt mit einem Verantwortlichen in der Versionskontrolle.
- [ ] Schreibende Sessions laufen mit ausgeschaltetem Auto-Commit und einem Row Cap, und Guarded Settings lassen sich innerhalb einer Session nicht ändern.

- [ ] Ein JIT-Schreibvorgang nennt eine geprüfte Korrektur, und der Runner lehnt jede Anweisungsmenge ab, deren Hash von `approved_hash` abweicht.
- [ ] Änderungen der Eigentümerverknüpfung und Schreibvorgänge in Restricted-Spalten gehen in jeder Lane auf den Produktpfad.
- [ ] Schemaänderungen laufen als `ledgerly_migrator` aus der Pipeline, und keine Person hält diesen Login.
- [ ] Der Export-Schritt läuft mit einer engen Rolle, und seine Datei hat einen benannten Verantwortlichen.

Zurechenbarkeit

- [ ] Jeder Login entspricht genau einer Person oder genau einer Workload-Instanz, und das Datenbank-Log zeigt es.
- [ ] Jeder Schreib- und jeder erhöhte Session-Datensatz trägt den Antrag, den Genehmiger, den Fingerprint und die zurückgegebenen Zeilen, und die Literale sind per Redaction entfernt.
- [ ] Der Audit-Speicher liegt in einem Konto, in das die Datenbankadministratoren und `ledgerly_access_admin` nicht schreiben können, und `verifyChain` besteht.
- [ ] Eine Lücke in der Sequenz, eine Gateway-Session ohne Datenbank-Session oder das Umgekehrte löst einen Befund aus.
- [ ] `evaluateAlertRules` und `runAccessReview` liefen im letzten Zyklus, und die Widerrufe, die sie vorgeschlagen haben, wurden umgesetzt oder mit Begründung akzeptiert.
- [ ] `containCompromise` wurde in einem Game Day eingeübt, und die Abfrage zum Auswirkungsbereich wurde aus der Aufzeichnung beantwortet.

Kultur: Der sichere Pfad muss der einfache Pfad sein

Jede Kontrolle des Buches ist eine Last für die Person, die sie verwendet, und diese Person entscheidet in der schlimmsten Nacht der Woche, ob sie sie trägt. Eine langsame oder oft ausfallende Kontrolle wird umgangen, weil der Entwurf den sicheren Weg zum schweren gemacht hat, und ein Incident wartet nicht. Die nützliche Antwort ist, die Umgehung als Mangel im Entwurf zu behandeln.

Ich würde eine Regel in die Mitte der Kultur stellen und messbar machen:

```
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
```

Zwei Gewohnheiten stützen die Regel. Zählen Sie, wie oft die Break-Glass-Rolle deklariert wird, und fragen Sie jedes Mal, ob stattdessen ein JIT-Pfad oder ein Produkt-Feature hätte dienen sollen; eine steigende Zahl sagt, dass der normale Pfad zu langsam ist, und eine Zahl von null über ein Jahr kann bedeuten, dass die Rolle funktioniert oder dass niemand glaubt, dass sie genehmigt würde, was ein Game Day klärt. Und geben Sie Genehmigern etwas, das sie nachts beantworten können: Eine Genehmigung, die Stunden dauert, bringt Antragstellern bei, mehr zu verlangen, als sie brauchen, und eine, die immer innerhalb von Sekunden erteilt wird, bringt niemandem bei, sie zu lesen (Kapitel 8 behandelt die Checkliste des Genehmigers).

Abwägungen und Kosten

Ein Schlusskapitel, das nur empfiehlt, würde den Fehler wiederholen, vor dem das Buch warnt, deshalb folgt hier, was der gesamte Entwurf kostet. Das System in der Referenzarchitektur hat viele Teile, und jedes davon muss in dem Moment verfügbar sein, in dem jemand es braucht. Das Gateway, der Runner, die Pipeline, der Broker, der Audit-Speicher und die Review-Jobs haben jeweils einen Verantwortlichen, einen Rufbereitschaftspfad und einen Ausfallmodus. Ein kleines Platform-Team kann feststellen, dass es seine Zeit damit verbringt, die Kontrollen am Laufen zu halten, und das ist eine echte Last, die man der beseitigten Gefährdung gegenüberstellen muss.

Die Checkliste und die Leiter haben eigene Kosten. Eine vierteljährlich gelesene Checkliste wird zum Ritual, wenn Befunde nicht geschlossen werden, und eine Leiter wird zum Statusbericht, wenn Teams auf den Namen der Stufe optimieren; der Ausstiegstest ist das Gegenmittel, weil er billig zu wiederholen und schwer zu fälschen ist. Die Bewertung „stark“ sagt, wofür ein Pattern gebaut ist, nicht dass ein bestimmtes Deployment davon es tut.

Wann dieser Apparat des Kapitels nicht einzusetzen ist: Ein System mit einem einzigen Entwickler, einer Datenbank und ohne Anbieter braucht einen Eigentümer-Login, den niemand benutzt, eine schreibgeschützte Rolle, ein Backup, das sich wiederherstellen lässt, und ein einseitiges Inventar. Das Buch richtet sich an den Punkt, an dem die Zahl der Principals oder die Kosten eines Fehlers darüber hinauswachsen, und ich weiß nicht, wo dieser Punkt für Ihr Unternehmen liegt; er hängt davon ab, wie viele Personen die Daten erreichen können, wie viel davon Restricted ist und wie lange ein Fehler unsichtbar bleibt.

Häufige Fehler

- **Die Checkliste als Zertifikat behandeln.** Es ist verlockend, einen Durchgang abzuschließen und das Ergebnis abzulegen. Die Checkliste zeigt, wo am Tag des Lesens der Nachweis fehlt. Wiederholen Sie sie nach Zeitplan, und halten Sie die Befunde offen, bis jeder gelöst oder mit Begründung und Datum akzeptiert ist.
- **Das Pattern ganz rechts zuerst bauen.** Ein anweisungsvermittelnder Executor ist attraktiv, weil er nach der meisten Kontrolle aussieht. Ohne das Inventar und eigene Principals darunter vermittelt er die Anweisungen eines gemeinsam genutzten Logins. Steigen Sie die Leiter der Reihe nach hinauf, oder schreiben Sie auf, warum Ihnen eine andere Reihenfolge besser passt.
- **Eine zweite Tür für Notfälle hinzufügen.** Sie sieht nach Resilienz aus und baut das gemeinsame Passwort neu auf. Braucht ein Team eine, während das Gateway reift, braucht sie einen Verantwortlichen, einen Alarm bei Benutzung und eine Bedingung für die Stilllegung (Kapitel 8).
- **Eine Umgehung als Verhaltensproblem lesen.** Wenn jemand eine Kontrolle umgeht, hat der Entwurf das billiger gemacht als den vorge-

sehenen Pfad. Fragen Sie, was der vorgesehene Pfad in diesem Moment kostete, und ändern Sie das.

Zentrale Erkenntnisse

- Der Zugriff auf Produktionsdatenbanken ist ein Problem der Autorisierung, der Ausführung und der Zurechenbarkeit, und ein Passwort berührt nur das Erste davon, und nur an der Tür.
- Die Arbeit an den Credentials aus Kapitel 4 ist notwendig, weil die Identität die Aufzeichnung erst möglich macht, und sie reicht nicht aus, weil sie nichts darüber sagt, was ein Principal tun darf oder was eine Anweisung bewirkt.
- Wählen Sie pro Aufgabe zwischen Direktzugriff, Bastion, Access-Gateway, anweisungsvermittelndem Executor und Produktpfad, mit dem günstigsten Pattern, das die Säulen, die die Aufgabe gefährdet, noch belegt.
- Eine Reifegradstufe ist abgeschlossen, wenn ihr Ausstiegstest besteht, und ein Test, den jemand wiederholen kann, ist die einzige Form von „fertig“, die die Drift überlebt.
- Jede Zeile der Checkliste ist eine Beobachtung eines Systems, eines Logs oder einer Tabelle; ein „nein“ ist ein Befund mit Verantwortlichem und Datum, nie ein Urteil über eine Person.
- Der sichere Pfad muss der einfache Pfad sein, und jede Umgehung ist eine Information darüber, wo der Entwurf den vorgesehenen Weg teuer gemacht hat.
- Kontrollen verringern den Auswirkungsbereich und machen Verhalten sichtbar, und sie kosten Verfügbarkeit, Aufmerksamkeit und Wartezeit; benennen Sie diese Kosten also, bevor Sie sie eingehen, und noch einmal, wenn sie anfallen.

Alex beantwortet Robins Frage am nächsten Morgen mit einem Test statt mit einer Verteidigung: Ein Teil bleibt, wenn es einen Teil des Modells liefert, den nichts anderes liefert, und sein Ausstiegstest besteht. Der Fragebogen wird wiederkommen, von einem anderen Kunden oder einer anderen Prüfstelle, und dieselben zwei Zeilen werden darauf stehen. Was das Buch Ihnen mitgeben kann, ist ein System, das die erste Zeile aus einem

Dokument beantwortet, die zweite aus einer Aufzeichnung und der diensthabenden Person ermöglicht, das Richtige zu tun, ohne heroische Anstrengung.

Anhang

Dieser Anhang ist Referenzmaterial zu den drei Säulen, aus denen das Buch argumentiert: Autorisierung, Ausführung und Zurechenbarkeit. Er besteht aus sechs Teilen: einem Glossar, einer Kurzreferenz der in Kapitel 16 bewerteten Patterns, der deny-/gate-/allow-Matrix aus Kapitel 9, einer Zuordnung von Fragen der Prüfer zu Artefakten und zwei Vorlagen, einer für das Zugriffs-Review aus Kapitel 14 und einer für das Post-Incident-Review aus Kapitel 15.

Nichts davon ist neu. Wo eine Tabelle eine aus einem Kapitel wiederholt, sind die Zellen die des Kapitels, und wenn beide voneinander abweichen, gilt das Kapitel. Die Auslegung eines Regelwerks liegt bei Ihren Rechts- und Compliance-Verantwortlichen; die Zuordnung listet technische Artefakte auf und erhebt keinen weiteren Anspruch.

A. Glossar

Die Begriffe sind alphabetisch geordnet, jeweils mit dem Kapitel, das sie aufbaut. Dies ist die abschließende Referenz für das Vokabular des Buches: Jeder Eintrag ist die Kurzform, und das genannte Kapitel enthält die vollständige Darstellung.

Begriff	Bedeutung
Abfrageanalyse	Die Prüfung einer Anweisung, bevor sie läuft: ihre Art, die Tabellen und Spalten, die sie berührt, ob ein Schreibvorgang einen begrenzten Row Guard hat, und die erwartete Zeilenzahl. Sie verringert Fehler und ist nicht die Autorisierungsgrenze (Kapitel 9).
access_audit_record	Die Einträge, die für jede Anweisung geschrieben werden, einer der Art intent und einer der Art outcome, mit Principal, Login, Antrag, `fingerprint` der Anweisung, berührten Tabellen und Spalten und, beim Outcome, Zeilenzahlen und Ergebnis. Er ist der Teil Aufzeichnung des Modells (Kapitel 14).
Access-Gateway	Ein identitätsbewusster Vermittler, der einen Menschen authentifiziert, ein kurzlebiges Credential ausstellt und die Session aufzeichnet. Weder eine Bastion noch ein Datenbank-Proxy (Kapitel 7).

Begriff	Bedeutung
Ad-hoc-Abfrage	Eine Abfrage, die ein Mensch schreibt und einmal ausführt, außerhalb jeder Anwendung (Kapitel 10).
Allowlist, Denylist	Eine Liste dessen, was erlaubt ist, und eine Liste dessen, was abgelehnt wird. Nie „Whitelist“ oder „Blacklist“ (Kapitel 7).
Analytics-Kopie	Der Speicher `ledgerly_analytics`. Er ist eine Kopie der Produktionsdaten, kein Replikat (Kapitel 6).
Änderung der Eigentümerverknüpfung	Ein `UPDATE` einer Spalte, die eine Zeile mit dem Kunden verknüpft, dem sie gehört, etwa `customer_id` in `bank_accounts`. Gate, und sie wird auf den Produktpfad geleitet (Kapitel 9).
an die Anweisung gebundene Genehmigung	Eine Genehmigung, die an genau die Anweisungsmenge geknüpft ist, die laufen wird, identifiziert durch `approved_hash`, und nicht nur an ein Zeitfenster oder eine Menge von Privilegien (Kapitel 8, 9, 10).
Approved Hash	Der `approved_hash` eines Zugriffsantrags: der Hash der genehmigten Anweisungsmenge, entnommen aus dem Merge-Eintrag der geprüften Korrektur oder dem genehmigten Antragseintrag, nie aus der Eingabe des Runners (Kapitel 9, 10).
Audit-Log	Die Aufzeichnung, wer was mit welchen Daten getan hat, aufbewahrt dort, wo der Handelnde sie nicht ändern kann. Die Tabelle `audit_events` ist die fachliche Aufzeichnung der Anwendung, nicht das Audit-Log (Kapitel 14).
Aufzeichnung	Der Teil des Modells, der fragt, welcher Nachweis bleibt und ob er die Anweisung mit der Person, dem Antrag und dem Ergebnis verbindet (Kapitel 14).
Ausführung	Wie eine Anweisung die Daten erreicht: wer oder was sie einreicht, ob etwas sie vor dem Lauf prüft, was ihre Wirkung begrenzt und wie sie mit einem Commit abgeschlossen wird (Kapitel 1, 9).
Auswirkungsbereich	Wie viel betroffen sein kann, wenn ein Principal, ein Credential oder eine Aktion schiefgeht. Immer bezogen auf eine bestimmte Sache angegeben (Kapitel 1, 2).
Automatisierungs-Principal	Ein Principal, der keine Person ist: ein Job, ein Runbook, ein ChatOps-Command, ein KI-Agent. Er hat eine Workload Identity und einen schreibgeschützten Standard, und die Aufzeichnung nennt sowohl ihn als auch die Person, die ihn ausgelöst hat (Kapitel 9).
Bastion	Ein gehärteter Host, der der einzige Netzwerkpfad zu einem geschützten System ist. Er leitet Verkehr weiter und versteht das Datenbankprotokoll nicht (Kapitel 7).
Break-Glass	Notfallzugriff mit umfangreicher Protokollierung und verpflichtendem Review. Die Rolle ist `human_breakglass`, und der Antrag wird selbst erklärt (Kapitel 8).

Begriff	Bedeutung
Credential	Das, was ein Principal zur Authentifizierung vorlegt: ein Passwort, ein Zertifikat, ein Token. Ein Credential ist eine Art von Secret (Kapitel 4).
Datenbank-Proxy	Ein protokollbewusster Vermittler, der Datenbankverbindungen bündelt, weiterleitet oder filtert. Er heißt Pooler, wenn es nur um das Pooling geht (Kapitel 5, 7).
Datenklassen	Die vier benannten Klassen von Daten: **Restricted** (eine Offenlegung lässt sich kaum rückgängig machen, etwa `iban` und `token_value`), **Confidential** (personenbezogene oder geschäftliche Daten über Kunden und Kontakte), **Internal** (betriebliche Daten ohne Personenbezug), **Public** (unbedenklich zu veröffentlichen). Als Klassennamen großgeschrieben (Kapitel 2, 12).
Datenklassifizierung	Das Einordnen von Daten in die benannten Klassen, die die Regeln für den Umgang bestimmen (Kapitel 12).
Datenkopie	Jeder Ort, an dem Produktionsdaten dupliziert wurden: Replikate, die Analytics-Kopie, Backups, Dumps, Exporte, Staging-Daten (Kapitel 6, 13).
Datenkorrektur	Eine Änderung an Produktionsdaten durch eine Person außerhalb des normalen Anwendungspfads, um einen Zustand zu reparieren, den die Anwendung nicht reparieren kann (Kapitel 10).
Dauerzugriff	Zugriff, der durchgehend besteht, ob er genutzt wird oder nicht (Kapitel 1).
deny / gate / allow	Die Antworten der Ausführungsrichtlinie, immer in dieser Reihenfolge. **Deny** läuft nie über diesen Pfad. **Gate** läuft erst nach zusätzlicher Genehmigung oder nur über den Produktpfad. **Allow** läuft innerhalb der Grants der Session (Kapitel 9).
dynamisches Credential	Ein kurzlebiges Credential, das auf Anfrage für genau einen Principal erzeugt wird, sodass zwei Anfragen nie denselben Wert erhalten (Kapitel 4).
Eindämmung	Das Beenden des Zugriffs eines Principals in fester Reihenfolge: Grants widerrufen, Sessions schließen, Credentials rotieren, den Schaden eingrenzen, aufzeichnen und benachrichtigen (Kapitel 15).
Grant	Das Erteilen eines Privilegs an eine Rolle, einer Rolle an einen Login oder eine andere Rolle oder, nur für JIT, eines Privilegs auf benannten Spalten direkt an einen Session-Login. Auch der daraus entstehende Eintrag (Kapitel 3).
Grant auf Spaltenebene	Ein Grant eines Privilegs auf benannten Spalten einer Tabelle. Wo die Engine ihn nicht kennt, liegen Restricted-Spalten in separaten Tabellen außerhalb der Obergrenze, und JIT-Grants fallen auf Tabellenebene zurück (Kapitel 3, 8).

Begriff	Bedeutung
Guarded Settings	Session-Einstellungen, die ein Principal innerhalb einer Session nicht ändern kann: Statement-Timeout, Row Cap, Auto-Commit-Modus, schreibgeschützter Standard, Rolle. Nur dort als nicht überschreibbar bezeichnet, wo ein Mechanismus das durchsetzt (Kapitel 5, 9).
Hash-Kette	Ein Aufzeichnungsformat, in dem jeder Eintrag den Hash des vorherigen trägt, sodass ein geänderter oder entfernter Eintrag `verifyChain` bricht. Sie liefert einen Nachweis der Manipulation und verhindert sie nicht (Kapitel 14).
Identität	Die verifizierte Aussage, dass ein Principal ist, wer er zu sein behauptet. Eine Rolle ist keine Identität (Kapitel 1).
Just-in-Time-Zugriff (JIT)	Zugriff, der für eine bestimmte Aufgabe und eine begrenzte Zeit erteilt wird (Kapitel 8).
kurzlebiges Credential	Ein Credential, das innerhalb einer kurzen, festgelegten Zeit von selbst abläuft (Kapitel 4).
Login	Das Konto auf Datenbankebene, mit dem sich ein Principal verbindet (Kapitel 1, 3).
Maskierung	Das Ersetzen oder Verschleiern von Werten, sodass der echte Wert nicht sichtbar ist, die Form aber nutzbar bleibt. Statische Maskierung ändert eine Kopie, dynamische Maskierung ändert, was eine Abfrage zurückgibt (Kapitel 12).
Migration	Eine versionierte Schemaänderung, die von `ledgerly_migrator` ausgeführt wird. „Schemaänderung“ ist der allgemeine Gedanke, „Migration“ das Artefakt (Kapitel 11).
minimale Rechte	Nur das erteilen, was eine Aufgabe braucht, und nur so lange, wie sie es braucht, wobei die Kosten dafür mitgezählt werden (Kapitel 1, 3).
Minimierung	Daten, die niemand braucht, nicht erheben, nicht kopieren und nicht aufbewahren (Kapitel 12).
Obergrenze	Das Meiste, was ein JIT-Antrag verlangen darf, festgelegt durch `human_jit_write`. Nie eine Restricted-Spalte (Kapitel 3, 8).
PII	Personenbezogene Daten (Personally Identifiable Information). „Sensible Daten“ ist der Oberbegriff für Restricted- und Confidential-Daten (Kapitel 2).
Primary, Replikat	Die beschreibbare Datenbank und ihre Lesekopien. Die Analytics-Kopie ist kein Replikat (Kapitel 6).
Principal	Alles, was sich authentifiziert: eine Person, ein Service, ein Job, eine Integration eines Anbieters (Kapitel 1).
Privileg	Eine bestimmte Sache, die ein Grant erlaubt, etwa `SELECT` auf einer Tabelle (Kapitel 1).

Begriff	Bedeutung
Produktpfad	Eine festgelegte Operation in einem Tool, etwa wenn `admin-console` den `payments-service` bittet, eine Restricted-Spalte zu ändern, ohne SQL für die Person (Kapitel 9, 10).
Redaction	Das vollständige Entfernen eines Werts aus einer Ausgabe oder einem Log (Kapitel 12).
Rolle	Ein benanntes Bündel von Privilegien in der Datenbank. Eine Rolle ist nie eine Person (Kapitel 3).
Runbook	Eine schriftlich festgehaltene, befolgte Prozedur für eine wiederholbare Aufgabe (Kapitel 10).
Runner	Der geprüfte Runner für Datenkorrekturen. Er führt eine Anweisungsmenge nur aus, wenn ihr Hash `approved_hash` entspricht, und das Gateway leitet JIT-Schreibvorgänge nur von ihm weiter (Kapitel 10).
Secret	Jeder Wert, der bei Offenlegung Zugriff gewährt (Kapitel 4).
Seitentür	Eine Datenkopie oder ein Exportpfad, der die Zugriffskontrollen des Primary umgeht (Kapitel 6, 13).
Session	Eine authentifizierte Verbindung oder eine vom Gateway vermittelte Interaktion eines Principals (Kapitel 5, 7).
Session Handout	Ein Modus, in dem das Gateway einem Principal einen Session-Login und eine Verbindung aushändigt und der eigene Client des Principals die Anweisungen sendet. Die Grants des Logins begrenzen alles (Kapitel 9).
Sicherheitstheater	Eine Kontrolle, die Reibung erzeugt, ohne die Gefährdung zu verringern (Kapitel 2, 8).
statisches Credential	Ein Credential, das gleich bleibt, bis jemand es ändert (Kapitel 4).
Tokenisierung	Das Ersetzen eines Werts durch einen Stellvertreter, den nur ein kontrollierter Service zurückabbilden kann. Sie ist weder Verschlüsselung noch Maskierung (Kapitel 12).
unabhängig von der Datenbank-Engine	Das Wort des Buches für Notation und Ratschläge, die zu keiner Datenbank-Engine gehören (Vorwort).
vermittelte Ausführung	Ein Modus, in dem der Principal nie eine Verbindung hält: Er reicht eine Anweisung ein, das Gateway oder ein Runner analysiert sie, und der Runner führt sie aus (Kapitel 9).
Vier-Augen-Prinzip	Genehmigung, bei der eine zweite Person prüft, bevor eine riskante Aktion läuft. Es sagt, wer genehmigt, nicht, was genehmigt wird (Kapitel 8).

Begriff	Bedeutung
Widerruf, Rotation	Ein Widerruf lässt ein Credential oder einen Grant sofort wirkungslos werden. Eine Rotation ersetzt ein Credential nach Zeitplan oder bei Verdacht und schließt bereits geöffnete Sessions nicht (Kapitel 4, 15).
Workload Identity	Eine Identität, die die Plattform einer laufenden Workload gibt, damit sie sich ohne gespeichertes Passwort authentifizieren kann (Kapitel 4).
Zugriffs-Review	Eine geplante Prüfung, ob der Zugriff jedes Principals noch gebraucht wird. Sie vergleicht, was erteilt wurde, mit dem, was genutzt wurde, und eine benannte Person entscheidet (Kapitel 14).

Mehrere dieser Paare werden verwechselt, und eine kurze Regel klärt jedes, bevor ein Streit entsteht.

Wenn die Frage lautet	Verwenden	Nicht
Ein Host, der Netzwerkverkehr weiterleitet, ohne Kenntnis von Anweisungen	Bastion	Access-Gateway
Ein Vermittler, der Session-Logins pro Person ausstellt und Sessions aufzeichnet	Access-Gateway	Datenbank-Proxy
Verbindungen bündeln oder weiterleiten, nichts über Personen	Pooler oder Datenbank-Proxy	Access-Gateway
Der eigene Client der Person sendet Anweisungen über eine Verbindung, die das Gateway ihr gegeben hat	Session Handout	vermittelte Ausführung
Die Person reicht eine Anweisung ein und hält nie eine Verbindung	vermittelte Ausführung	Session Handout
Verhindern, dass ein Credential sofort weiter funktioniert	Widerruf	Rotation
Wer eine riskante Aktion genehmigt	Vier-Augen-Prinzip	an die Anweisung gebundene Genehmigung
Was genau genehmigt wurde, per Hash	an die Anweisung gebundene Genehmigung	Vier-Augen-Prinzip
Die fachliche Aufzeichnung der Anwendung	`audit_events`	Audit-Log

B. Kurzreferenz der Patterns

Kapitel 16 bewertet fünf Patterns anhand der drei Säulen. Die Bewertungen sind das Urteil des Buches und keine Messung: „stark“ heißt, das Pattern ist darauf ausgelegt, die Säule nachzuweisen, „teilweise“, dass es einen Teil davon oder nur mit einer Ergänzung nachweist, „keine“, dass es dazu schweigt. Die Spalte Kosten ist die aus Kapitel 16, gekürzt.

Pattern	Was es löst	Kosten	Autorisierung	Ausführung	Zurechenbarkeit	Kapitel
Direktzugriff (Client zur Datenbank)	Ein Login existiert, und seine Grants waren, was sie waren	Nichts im Aufbau; jede Kostenstelle kommt später, als eine Woche Gräberei	Teilweise: nur Grants, die Reichweite ist, was das Netzwerk erlaubt	Keine	Bestenfalls teilweise: Logins pro Person, wenn jemand sie bereitstellt, sonst keine	1
Bastion	Ein kontrollierter Host, eine Stelle, um den Netzwerkpfad abzuschalten	Ein Host, der mit Patches versorgt und verfügbar gehalten werden muss	Teilweise: verengt den Netzwerkpfad	Keine: Sie leitet weiter und kann ein `SELECT` nicht von einem `DROP` unterscheiden	Teilweise: wer sich am Host angemeldet hat, nicht was lief	7
Session-Proxy oder Access-Gateway	Eine benannte Person hielt einen bestimmten Session-Login, zwischen zwei Zeitstempeln, unter bestimmten Grants	Ein neues System, das verfügbar sein muss, wenn jemand es braucht, und der Genehmigungspfad dazu	Stark: Session-Logins pro Person, Ablauf, Grants pro Antrag	Teilweise: begrenzt durch Grants und Guarded Settings, Anweisungen werden frei getippt	Stark bei wer und wann, teilweise bei was	7, 8
Anweisungsvermittelnder Executor	Genau diese Anweisungsmenge wurde per Hash genehmigt, geprüft, begrenzt und gezählt	Jede Aufgabe wartet auf ein Review, nichts ist interaktiv, jemand baut und betreibt den Runner	Stark	Stark: vor der Ausführung geprüft, an einen Approved Hash gebunden	Stark: Anweisung, Genehmiger und Ergebnis in einer Aufzeichnung	9, 10
Produktpfad (kein SQL)	Eine festgelegte Operation lief, für eine Rolle in einem Tool	Ein Feature pro Aufgabe und der langsamste Änderungszyklus der fünf	Stark: die Operation ist die Berechtigung	Stark: die Operation ist festgelegt	Teilweise: die Person steht im Prüfpfad des Tools, das Datenbank-Log zeigt den Service-Login	9, 10

Die Wahl unter ihnen ist eine Entscheidung pro Aufgabe, und die Regel ist die aus Kapitel 16: das günstigste Pattern, das die Säulen, die die Aufgabe gefährdet, noch nachweist.

```
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. Matrix gefährlicher Operationen

Die folgende Matrix ist die Ausführungsrichtlinie aus Kapitel 9, wiederholt mit unveränderten Zellen. Die Zeilen sind die Anweisungsklassen, die der Klassifikator erzeugt. Die Spalten sind die Lanes, in denen eine Anweisung ankommen kann. Jede Zelle hat eine von drei Antworten: deny (läuft nie über diesen Pfad), gate (braucht eine Genehmigung, die sie noch nicht hat, oder gehört auf eine andere Route), allow (läuft innerhalb der Grenzen der Schleife, und wo eine Zelle „gebunden“ sagt, nur innerhalb der Grenze aus der genannten Quelle).

Anweisungsklasse	Lese-Session für Menschen	JIT-Schreibzugriff	Break-Glass	Runner für Datenkorrekturen	Migration
Lesen, innerhalb des Row Cap	Allow	Allow	Allow	Allow	Allow

Anweisungsklasse	Lese-Session für Menschen	JIT-Schreibzugriff	Break-Glass	Runner für Datenkorrekturen	Migration
Begrenzter Schreibvorgang, innerhalb der genehmigten Anzahl	Deny	Allow, gebunden an die Korrektur (nur Runner)	Allow, gebunden an einen Row Cap (Notiz erforderlich)	Allow, gebunden an den Approved Hash und die Anzahl	Allow, gebunden an die Backfill-Regel
Unbegrenzter Schreibvorgang	Deny	Deny	Deny	Deny	Gate: benannter Genehmiger
Änderung der Eigentümerverknüpfung	Deny	Gate: Produktpfad	Gate: Produktpfad	Gate: Produktpfad	Gate: benannter Genehmiger
Schreiben in eine Restricted-Spalte	Deny	Gate: Produktpfad	Gate: Produktpfad	Gate: Produktpfad	Gate: benannter Genehmiger
Export	Gate: nach Stufe	Deny	Deny	Deny	Gate: benannter Genehmiger
Schemaänderung (DDL)	Deny	Deny	Deny	Deny	Allow: geprüfte Lane
Änderung von Privilegien	Deny	Deny	Deny	Deny	Gate: Plattform-Review
Schreiben in `ledger_entries` oder `audit_events`	Deny	Deny	Deny	Deny	Gate: Plattform-Review

Drei Leseregeln ersparen die meisten Nachschlagevorgänge.

- **Ein Schreibvorgang in den Händen einer Person ist verweigert oder gebunden.** Die Lese-Session für Menschen hat keinen Schreibpfad, JIT-Schreibvorgänge kommen nur vom Runner, und Break-Glass

ist die einzige Lane, in der eine Person direkt schreibt, unter einem Row Cap und mit einer Notiz.

- **Ein Gate nennt, wohin die Anweisung als Nächstes geht.** „Produktpfad“ heißt, `admin-console` bittet `payments-service`, eine zweite Person genehmigt, und die Änderung wird in `audit_events` aufgezeichnet. „Benannter Genehmiger“ und „Platform-Review“ heißen, dass die Migrations-Lane die Anweisung für eine Person markiert, die zustimmen muss.
- **Die Matrix ist nicht die Autorisierungsgrenze.** Die Abfrageanalyse verringert Fehler. Die Grants des Logins bleiben die Grenze, und eine Anweisung, die der Klassifikator falsch liest, wird trotzdem von ihnen begrenzt (Kapitel 9).

Die Entscheidung für eine einzelne Anweisung, als Regel geschrieben:

```
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. Compliance-Zuordnung

Regelwerke für Datenschutz und Zahlungskarten stellen tendenziell eine Familie von Fragen zum Zugriff. Ein Prüfer, der unter einem dieser Regelwerke arbeitet, will meist sehen, dass eine Kontrolle existiert, dass sie lief und dass ihre Ergebnisse angesehen wurden. Diese Zuordnung listet die Fragetypen auf, die Kontrolle, die jeweils antwortet, das Artefakt, das sie hinterlässt, und das Kapitel, das sie aufbaut. Sie ist die Tabelle aus Kapitel 14, wiederholt, und behauptet nicht, dass irgendeine Kontrolle irgendeine Verpflichtung erfüllt.

Frage	Kontrolle	Artefakt	Kapitel
Wer kann Produktionsdaten erreichen, und auf welchem Pfad?	Gateway als einzige Tür, Rollenschema als Code, Drift-Prüfung	Liste der Netzwerkregeln, Rollendefinitionen, Drift-Bericht	3, 7
Wer hat eine bestimmte Anweisung ausgeführt, mit welchen Grants und welcher Genehmigung?	`access_audit_record` mit Request-ID und Genehmiger	Die Aufzeichnung und der Bericht von `verifyChain`	14
Wie lange dauert erhöhter Zugriff, und wer hat ihn genehmigt?	JIT-Antrag mit Genehmiger und `expires_at`, Widerrufs-Job	Antragseinträge, Widerrufs-Log, Prüfung auf überfällige Grants	8
Wie werden sensible Daten identifiziert und begrenzt?	Klassifizierung im Schema, Maskierung, zurückgehaltene Spalten	Der Katalog, die Verifikation des maskierten Ladens	12
Wo liegen Kopien von Produktionsdaten, und wem gehört jede?	Register mit Verantwortlichem und Enddatum, Sweeper	Register, Sweeper-Log	6, 13
Wer ändert das Schema, und wer hat es geprüft?	Migrations-Pipeline, geschützter Branch, externer Starteintrag	Migrationseinträge, abgeglichen mit Commits und Audit-Einträgen	11
Wird der Zugriff noch gebraucht?	Zugriffs-Review mit benannten Reviewern	Abgeschlossenes Review mit Entscheidungen	14
Was geschieht im Notfall?	Break-Glass mit Alarm und verpflichtendem Review	Review-Einträge, Alarmhistorie	8, 15
Wer verwaltet den Zugriff selbst?	Logins von `ledgerly_access_admin` als separate Producer	Ihre Streams und der Vergleich von Katalog und Aufzeichnung	3, 14

Eine Kontrolle, die existiert, aber nie lief, oder die lief und nie angesehen wurde, beantwortet nur ein Drittel der Frage. Die folgende Prüfung ist die, die man ausführt, bevor man jemandem, der danach gefragt hat, ein Artefakt übergibt.

```
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. Vorlage für Zugriffs-Reviews

Dies ist die Vorlage hinter `runAccessReview` aus Kapitel 14. Kopieren Sie sie für jeden Review-Zyklus. Das Review schlägt vor, und eine benannte Person entscheidet; niemand prüft seinen eigenen Zugriff, und der Zeitraum ist länger als der längste Zyklus der Workload, die geprüft wird (ein einmonatiges Review würde vorschlagen, einen Grant zu widerrufen, den ein Quartalsbericht in zwei von drei Monaten braucht).

Kopfbereich

```
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?>
```

Teil 1: erklärte Grants gegenüber der Nutzung

Grant (Rolle, Privileg, Tabelle) oder Login	Verantwortlicher	Letzte Nutzung in der Aufzeichnung	Befund	Entscheidung	Entschieden von	Erledigt am
`<role>`, `<privilege>`, `<table>`	`<named principal>`	`<date or none>`	ungenutzt / kein Verantwortlicher / Nutzung enger als der Grant / in Ordnung	behalten / auf `<privilege>` verengen / widerrufen / Verantwortlichen zuweisen	`<reviewer>`	`<date>`

Gleichen Sie die Nutzung nach Privileg ab und nicht nur nach Tabelle. Eine Rolle, die jede Nacht liest und nie aktualisiert hat, ist ein Kandidat dafür, von `S, U` auf `S` verengt zu werden. Beziehen Sie die Zuordnungen von Gruppen des Identity-Providers zu Rollen und die Logins von `ledgerly_access_admin` ein.

Teil 2: Dinge, die keine Grants sind

Element	Frage	Antwort	Verantwortlicher	Entscheidung
Register der Datenkopien (Kapitel 13)	Hat jede Kopie noch einen Verantwortlichen und ein künftiges Enddatum?	`<yes / no, which>`	`<name>`	`<keep / end by date>`
Wer JIT-Anträge genehmigen darf	Hat jeder Genehmiger noch eine Rolle, die das rechtfertigt?	`<yes / no, which>`	`<name>`	`<keep / remove>`
Wer den Audit-Speicher lesen kann	Braucht ihn jeder Leser noch?	`<yes / no, which>`	`<name>`	`<keep / remove>`
Automatisierungs-Principals (Jobs, Runbooks, KI-Agenten)	Läuft jeder noch, mit einer Workload Identity und einem schreibgeschützten Standard?	`<yes / no, which>`	`<name>`	`<keep / retire>`

Teil 3: Break-Glass-Nutzungen im Zeitraum

Request-ID	Genutzt von	Incident oder Grund	Geprüft von	Review-Datum	Befund zum System
`<id>`	`<principal>`	`<reason>`	`<other than user>`	`<date or MISSING>`	`<for example: missing path>`

Eine Zeile ohne Review-Datum ist selbst ein Befund.

Abschlussregeln

```
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. Vorlage für Post-Incident-Reviews

Diese Vorlage folgt Kapitel 15. Sie ist schuldzuweisungsfrei: Jede Frage betrifft das System, das eine Aktion zuließ oder unsichtbar machte, und keine fragt, wer es hätte besser wissen sollen. Füllen Sie sie aus der Aufzeichnung aus, nicht aus dem Gedächtnis. Wo die Aufzeichnung eine Frage nicht beantworten kann, schreiben Sie das, denn die Lücke ist ein Befund. Zeiten sind die, die die Aufzeichnung zeigt; schätzen Sie nicht.

1. Zusammenfassung

```
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. Eindämmung, in Reihenfolge

Schritt	Erledigt um	Ergebnis	Anmerkungen dazu, was im Weg stand
Grants widerrufen	`<timestamp>`	erledigt / fehlgeschlagen	
Sessions schließen	`<timestamp>`	erledigt / fehlgeschlagen	
Credentials rotieren	`<timestamp>`	erledigt / fehlgeschlagen	
Den Schaden anhand des Audit-Logs eingrenzen	`<timestamp>`	erledigt / fehlgeschlagen	

Schritt	Erledigt um	Ergebnis	Anmerkungen dazu, was im Weg stand
Aufzeichnen und benachrichtigen	`<timestamp>`	erledigt / fehlgeschlagen	

Fragen: Lief jeder Schritt, auch wenn ein früherer fehlschlug? Blieben Schritte unvollendet, und wurden sie nachverfolgt? War das Log erreichbar, und falls nicht, wurden Einträge in eine Warteschlange gestellt, ohne den Entzug des Zugriffs zu verzögern?

3. Auswirkungsbereich

```
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. Lebenszyklus des Incident-Zugriffs

Request-ID	Art	Beantragt	Erteilt	Erste Nutzung	Widerrufen	Geprüft von
`<id>`	jit / breakglass	`<time>`	`<time>`	`<time or none>`	`<time>`	`<other than user, or MISSING>`

Fragen: Blieb ein Grant über `expires_at` hinaus bestehen? Wurde Zugriff erteilt, den kein Antrag erklärt? Fand `closeIncident` offenen Zugriff, und wurde er geschlossen, bevor der Incident geschlossen wurde?

5. Schreibvorgänge während des Incidents

Schreibvorgang	Route (Runner mit beschleunigtem Review, Produktpfad, rohes Break-Glass)	Reviewer, der nicht der Autor war	Approved Hash stimmte überein
`<description>`	`<route>`	`<name>`	ja / nein

Frage: Wurde rohes Break-Glass für einen Schreibvorgang genutzt, wo der Runner ausgereicht hätte, und warum?

6. Zurückgestellte Fälle, falls sie eintraten

- **Gateway nicht verfügbar:** wie lange der Zugriff für Menschen gestoppt war, ob die Services weiterliefen und ob das Runbook zur Wiederherstellung wie geschrieben funktionierte.
- **Wiederherstellung während des Incidents:** ob die wiederhergestellte Kopie mit einem Verantwortlichen und einem Enddatum registriert und beendet wurde.
- **Langsame Genehmigungen in der Nacht:** welcher Antrag wartete, auf wen, und was die Regel für einen unbeantworteten Antrag bewirkte.

7. Befunde und Folgemaßnahmen

Befund zum System	Teil (Identität, Grant, Pfad, Ausführung, Aufzeichnung)	Änderung	Verantwortlicher	Endet am
`<for example: a path that should have existed>`	`<part>`	`<change>`	`<name>`	`<date>`

Schließen Sie mit einer Prüfung: Ist jeder Befund eine Änderung an einem Design mit einem Verantwortlichen und einem Enddatum und keine Erinnerung, vorsichtiger zu sein?

THE UNRELIABLE
SYSTEMS TRILOGY

I

ÜBER DAS BUCH

Der Zugriff auf Produktionsdatenbanken ist kein Problem der Credential-Verteilung; es ist ein Problem der Autorisierung, der Ausführung und der Zurechenbarkeit.

Ein besserer Passwort-Tresor ändert nur, wie ein Secret zu einem Menschen gelangt. Er entscheidet nicht, was dieser Mensch tun darf, was tatsächlich gegen die Daten ausgeführt wird oder wer danach dafür geradesteht. Dieses Buch begleitet Ledgerly von einem gemeinsam genutzten Admin-Passwort bis zu einem bewusst entworfenen Zugriffsmodell: Identitäten mit minimalen Rechten, Just-in-Time- und Break-Glass-Zugriff, vermittelte Ausführung für gefährliche Operationen und ein Prüfpfad, der sagen kann, wer was getan hat und warum. Die Patterns sind unabhängig von der Datenbank-Engine, und jedes wird mit den Kosten genannt, die es mit sich bringt.

ÜBER DEN AUTOR

Muhammet Şafak entwickelt seit 2008 Software. Über viele Jahre war Muhammet in den Bereichen Gesundheitswesen, Finanzen und Telekommunikation aktiv tätig und vertritt die Überzeugung, dass Entwicklung beim Problem ansetzen sollte, nicht bei der Technologie. Seit 2024 entwirft und entwickelt Muhammet hochfrequentierte Systeme, die durchgängig skalieren.

Muhammet bietet kleinen und mittelgroßen Softwareentwicklungsteams Mentoring an, leitet Open-Source-Projekte und hält Entscheidungen schriftlich fest – mit ihren Gründen und ihrem Preis.

Zuverlässige Systeme werden nicht aus zuverlässigen Teilen gebaut. Sie werden um die Teile herum gebaut, die ausfallen.

I ZUGRIFF

II EREIGNISSE

III AGENTEN