BE SOFTWARE ARCHITECTURE WISE · KAPITEL 7

Datenarchitektur

Daten überdauern Code, und in verteilten Systemen sind sie der am schwierigsten richtig zu gestaltende Teil. Dieses Kapitel behandelt, wem welche Daten gehören und warum eine gemeinsame Datenbank Services koppelt, wie man zwischen relationalen und anderen Datenbanken wählt, was Transaktionen und Isolationsebenen wirklich garantieren, wie Replikation und Partitionierung funktionieren und versagen, CQRS und Event-Sourcing, Caching, die Weiterentwicklung von Schemas ohne Ausfallzeit sowie Analytics, Streaming und Data Mesh. Es richtet sich an Entwicklungs- und Architekturfachleute, die die verteilten Stile aus Kapitel 6 bereits kennen und fundierte, gut begründbare Entscheidungen über Daten treffen müssen.

bewiseapps.com/de/software-architecture-wise/data-architecture/ · Version 2026-11-03. Nur Bildungsinformation, keine professionelle Beratung.

AUF EINEN BLICK

Das lernen Sie

- Erklären, warum eine gemeinsame Datenbank Services koppelt, und die Datenhoheit zuweisen, auch in Fällen, in denen mehrere Services dieselben Daten schreiben.
- Für jeden Datenbestand eine Datenbank nach Datenform, Zugriffsmustern, Konsistenzanforderungen, Skalierung, Betrieb und Fähigkeiten des Teams auswählen und die Kosten jedes zusätzlichen Speichers begründen.
- Die Isolationsebenen und die Anomalien, die sie zulassen, unterscheiden, darunter Lost Updates und Write Skew, und einen Read-Modify-Write-Pfad mit Sperren, Versionsprüfungen oder serialisierbaren Transaktionen absichern.
- Leader-Follower-, Multi-Leader- und Leaderless Replication erklären, ebenso die Anomalien durch Replikationsverzögerung und die Risiken von Failover und Split Brain.
- Ein Partitionierungsschema und einen Shard-Schlüssel anhand gemessener Volumina und Abfragen wählen und konsistentes Hashing, Hot Keys und Sekundärindizes erklären.
- Entscheiden, wann CQRS, Event-Sourcing und Caching ihre Kosten wert sind, und ihre typischen Fehlerbilder vermeiden.
- Schemaänderungen mit Kompatibilitätsregeln und dem Expand-and-Contract-Muster so planen, dass sie keine Ausfallzeit brauchen.
- Operative und analytische Datenarchitekturen unterscheiden: Data Warehouses, Data Lakes, Lakehouses, Stream Processing und Data Mesh.

Bevor Sie beginnen

Kapitel 6, besonders die Abschnitte über Microservices, ereignisgesteuerte Architektur, Konsistenz, CAP und PACELC sowie Sagas. Der Abschnitt über das Speichern von Daten in Kapitel 0 deckt die hier vorausgesetzten Datenbankgrundlagen ab.

Dauer

Etwa 3½ Stunden Lesezeit; 45 Minuten für den Test.

Schlüsselbegriffe

System of Record · Datenbank pro Service · polyglotte Persistenz · Isolationsebene · Snapshot Isolation · Write Skew · Replikationsverzögerung · konsistentes Hashing · CQRS · Event-Sourcing · Cache-Aside · Expand and Contract · Lakehouse · Data Mesh

Warum Daten der schwierige Teil sind

Code lässt sich in Monaten neu schreiben; Daten leben Jahrzehnte. Kundenkonten, Hauptbücher, Bestellungen und Krankengeschichten überdauern jedes Framework und jeden Service, der mit ihnen arbeitet, und sie zu migrieren ist langsam, riskant und teuer. Deshalb gehören die Entscheidungen in diesem Kapitel zu den bedeutsamsten, die ein Architekt trifft, im Sinne von Kapitel 1: Sie sind schwer rückgängig zu machen, weitreichend und entscheidend für Qualitätsmerkmale wie Verfügbarkeit, Performance und Änderbarkeit.

Kapitel 6 hat Systeme in Services aufgeteilt, die über ein Netzwerk kommunizieren. Code aufzuteilen ist die leichte Hälfte. Sobald es mehrere Services gibt, muss jemand entscheiden, wer welche Daten ändern darf, wie andere sie sehen, welche Garantien gelten, wenn viele Nutzer gleichzeitig handeln, und was geschieht, wenn Kopien der Daten voneinander abweichen. Das Kapitel folgt durchgehend zwei Beispielen: einer Essenslieferplattform mit ihrem abendlichen Ansturm, Kurierstandorten und Speisekarten und einer Onlinebank, bei der es inakzeptabel ist, auch nur eine einzige Zahlung zu verlieren oder doppelt zu zählen.

Daten besitzen, speichern und schützen

Die ersten vier Abschnitte behandeln die operativen Daten, auf denen Systeme laufen. Datenhoheit legt die wichtigste Regel fest: Jede wichtige Tatsache sollte genau ein System of Record und genau einen Schreiber haben. Der Abschnitt erklärt, warum eine gemeinsame Integrationsdatenbank Services koppelt, wie sauber ihr Code auch getrennt sein mag, wie man Fälle löst, in denen mehrere Services dieselben Daten schreiben wollen, wie man Daten liest, die einem nicht gehören, und wie Datenverträge gemeinsam genutzte Daten verlässlich machen.

Eine Datenbank wählen gibt einen Überblick über relationale, Dokument-, Key-Value-, Wide-Column-, Graph-, Zeitreihen-, Such-, Vektor- und Distributed-SQL-Speicher und begründet, warum die relationale Datenbank der richtige Standard für Systems of Record bleibt und spezialisierte Speicher sich ihren Platz verdienen müssen, indem sie eine Anforderung erfüllen, die der Standard nicht erfüllen kann. Transaktionen und Isolation blickt unter das vertraute Akronym ACID auf die Anomalien, die Nebenläufigkeit verursacht, etwa Lost Updates und Write Skew, darauf, was Isolationsebenen in PostgreSQL, MySQL, Oracle und SQL Server wirklich garantieren, und darauf, wann Sperren, Versionsprüfungen oder serialisierbare Transaktionen angebracht sind. Replikation und Partitionierung erklärt, wie Datenbanken Daten über Maschinen hinweg kopieren und aufteilen, was Replikationsverzögerung und Failover mit der Korrektheit machen und wie man einen Shard-Schlüssel anhand der Zahlen wählt.

Daten ableiten, cachen und weiterentwickeln

Die zweite Hälfte behandelt Daten, die aus den Systems of Record abgeleitet werden, und Veränderung über die Zeit. CQRS und Event-Sourcing trennt das Modell, das Änderungen annimmt, von den Modellen, die Abfragen beantworten, und zeigt, wie das Speichern einer Ereignishistorie statt des aktuellen Zustands Audits und zeitbezogene Abfragen ermöglicht, zu einem echten Preis an Komplexität. Caching behandelt die Muster, Ebenen und Fehlerbilder

des Vorhaltens von Kopien nahe am Ort ihres Bedarfs, vom Browser bis zur Datenbank, einschließlich der Stampedes, die ein System lahmlegen können, wenn ein beliebter Eintrag abläuft.

Schemaevolution befasst sich damit, dass sich jedes Schema, ob Datenbanktabellen, API-Payloads oder Events, im laufenden Betrieb ändern wird, und zeigt, wie Kompatibilitätsregeln und das Expand-and-Contract-Muster Änderungen ohne Ausfallzeit ermöglichen. Zum Schluss geht Analytics und Streaming vom Betreiben des Geschäfts zu seinem Verstehen über: Data Warehouses, Data Lakes und Lakehouses, Stream Processing und Data Mesh als Weg, analytische Daten entlang von Domänen zu organisieren.

Wie die Abschnitte zusammenhängen

Die Abschnitte bilden eine Kette. Die Datenhoheit entscheidet, wo jede Tatsache geschrieben wird. Die Wahl der Datenbank und ihre Transaktionsgarantien entscheiden, welche Regeln sich innerhalb dieses Eigentümers günstig durchsetzen lassen. Replikation und Partitionierung entscheiden, wie die Daten des Eigentümers Ausfälle überstehen und über eine Maschine hinauswachsen. Alles andere im Kapitel ist eine bewusste Kopie: Read Models, Caches, Suchindizes, Data Warehouses und Streams. Jede Kopie ist für einen bestimmten Zweck schneller oder bequemer, und jede ist um ein gewisses Maß veraltet. Die Schemaevolution sorgt dafür, dass sich der Eigentümer und alle seine Kopien ändern können, ohne einander zu beschädigen.

Das Kapitel setzt das Vokabular verteilter Systeme aus Kapitel 6 voraus, besonders Konsistenzmodelle, ereignisgesteuerte Architektur sowie Sagas und das Outbox-Muster. Es führt weiter zu Kapitel 8, das diese Daten schützt, und zu Kapitel 9, das die Systeme, die sie halten, schnell und verfügbar hält.

Der eine Gedanke, den Sie mitnehmen sollten

Für jedes wichtige Datum sollten Sie drei Fragen beantworten können: Wer ist sein einziger Schreiber, wo werden seine Regeln durchgesetzt, und wie veraltet darf jede Kopie davon sein? Die meisten Datenprobleme in realen Systemen, etwa Bestellungen, deren Status rückwärts springt, Berichte, die dem Hauptbuch widersprechen, Doppelbuchungen und Kunden, die ihre eigenen Änderungen nicht sehen, entstehen, weil eine dieser Fragen nicht bewusst beantwortet wurde. Die Techniken in diesem Kapitel sind Wege, sie gut zu beantworten, und jede von ihnen ist eine Abwägung zwischen Konsistenz, Verfügbarkeit, Performance, Kosten und dem Aufwand für den Betrieb des Systems.

Datenhoheit und Datenbank pro Service

BEVOR SIE DAS LESEN

Dieser Abschnitt baut auf:

- 0.5 Daten speichern
- 6.3 Microservices
- 6.4 Ereignisgesteuert

Begriffe von früher, die er verwendet: Saga, Eventual Consistency, Schema, Transaktion

DIE IDEE IN KÜRZE

Stellen Sie sich eine gemeinsame Pinnwand in einer Küche vor, auf der jeder die Bestellungen des Tages wegwischen und neu schreiben kann. Mit zwei Köchen funktioniert das. Mit zehn werden Bestellungen überschrieben, niemand weiß, wer was geändert hat, und eine Neugestaltung der Tafel erfordert die Zustimmung aller. Eine bessere Küche gibt jeder Station ihre eigene Tafel und ihre eigene Verantwortung: Der Grill schreibt die Grillbestellungen, und andere Stationen lesen sie oder bitten den Grill, sie zu ändern. Datenhoheit bedeutet, für jede Information festzulegen, welcher Teil eines Systems sie als einziger ändern darf.

WAS SIE WISSEN MÜSSEN

- Ein System of Record ist die maßgebliche Quelle für ein Datum; Datenhoheit bedeutet, dass genau eine Komponente es schreibt, seine Regeln durchsetzt, sein Schema verantwortet und es für andere veröffentlicht.
- Eine gemeinsame (Integrations-)Datenbank koppelt Services über Schema, Deployment, Semantik und Last zur Laufzeit; Fowler (2004) und die meisten Architekten raten davon zwischen unabhängig entwickelten Anwendungen ab.
- Datenbank pro Service hält die Daten jedes Service privat (eigene Tabellen, eigenes Schema oder eigener Server), um den Preis von Joins, Transaktionen und referenzieller Integrität über Services hinweg, die durch APIs, Events, Sagas und Reporting-Pipelines ersetzt werden müssen.
- Gemeinsame Datenhoheit lässt sich durch Aufteilen der Tabelle, eine kleine gemeinsame Datendomäne, Delegation an einen Eigentümer oder Zusammenlegen der Services auflösen, jeweils mit einer eigenen Abwägung (The Hard Parts, 2021).

- Daten, die einem nicht gehören, lassen sich über synchrone Aufrufe, Column Schema Replication, replizierte Caches oder Events lesen; jede Kopie ist schreibgeschützt und möglicherweise veraltet.
- Datenverträge legen Schema, Bedeutung, Qualität, Service-Levels und Verantwortung für Daten fest, die Teams miteinander teilen, und sind am nützlichsten, wenn sie automatisch geprüft werden.
- Für jede wichtige Tatsache kann ein solides Design ihren einzigen Schreiber benennen, sagen, wo ihre Regeln durchgesetzt werden, und wie veraltet jede Kopie sein darf.

Warum Datenhoheit die erste Datenfrage ist

Kapitel 6 hat Systeme in Services aufgeteilt, die unabhängig deployt, skaliert und geändert werden. Code lässt sich relativ leicht aufteilen, Daten nicht. Eine Bestellung verweist auf einen Kunden, einen Speisekartenartikel, ein Restaurant und einen Kurier, und jedes Team will einen Teil davon lesen und oft auch ändern. Die wichtigste Frage einer verteilten Datenarchitektur ist deshalb nicht, welche Datenbank man kauft, sondern **wem welche Daten gehören**: Welche Komponente darf sie ändern, welche setzt ihre Regeln durch, und wie bekommen alle anderen sie zu sehen?

Datenhoheit ist wichtig, weil Daten Code überdauern. Services werden alle paar Jahre neu geschrieben; Kundenkonten, Hauptbücher und Patientenakten werden jahrzehntelang aufbewahrt. Eine falsch geregelte Datenhoheit erzeugt die teure Art von Architekturschulden: zwei Services, die sich nicht getrennt releasen lassen, weil sie sich Tabellen teilen, ein Statusfeld, das drei Teams abwechselnd überschreiben, oder ein Monatsbericht, den niemand reproduzieren kann, weil die Zahlen aus fünf Kopien derselben Daten stammen, die voneinander abweichen.

Dieser Abschnitt nutzt zwei durchgehende Beispiele, die im ganzen Kapitel wiederkehren: eine Essenslieferplattform (Kunden, Restaurants, Kuriere, Bestellungen) und eine Onlinebank (Kunden, Konten, Zahlungen, ein Hauptbuch).

System of Record und Datenhoheit

Ein System of Record ist die maßgebliche Quelle für ein bestimmtes Datum: der Ort, dessen Wert gilt, wenn Kopien voneinander abweichen. In der Bank ist das zentrale Hauptbuch das System of Record für Kontostände; der Kontostand in der Mobile-App, der im Cache der Betrugserkennung und der in der Ladung des Data Warehouse von letzter Nacht sind alles Kopien. In der Essenslieferplattform ist der Speisekarten-Service des Restaurants das System of Record für Preise, obwohl der Suchindex, der Offline-Cache der Mobile-App und die Bestellpositionen alle Kopien davon halten.

Datenhoheit macht daraus eine Verantwortung. Die besitzende Komponente (und das Team dahinter) ist die einzige, die die Daten schreibt, ihre *Invarianten* (Regeln, die immer gelten müssen, etwa „ein Hauptbucheintrag wird nie bearbeitet, nur storniert“) durchsetzt, ihr Schema festlegt und sie über eine von ihr kontrollierte Schnittstelle für andere veröffentlicht. Datenhoheit ist eine Designentscheidung, keine Eigenschaft der Datenbank: Zwei Services können auf demselben Datenbankserver liegen und trotzdem eine klare Datenhoheit haben, und zwei Services mit getrennten Datenbanken können immer noch um dieselben Tatsachen streiten, wenn die Grenze zwischen ihnen schlecht gezogen ist.

Eine nützliche Regel aus *Software Architecture: The Hard Parts* (Neal Ford, Mark Richards, Pramod Sadalage und Zhamak Dehghani, 2021) lautet: Der Service, der in eine Tabelle schreibt, besitzt sie. Lesen lässt sich günstig teilen, Schreiben nicht, denn wer schreibt, muss die Regeln durchsetzen.

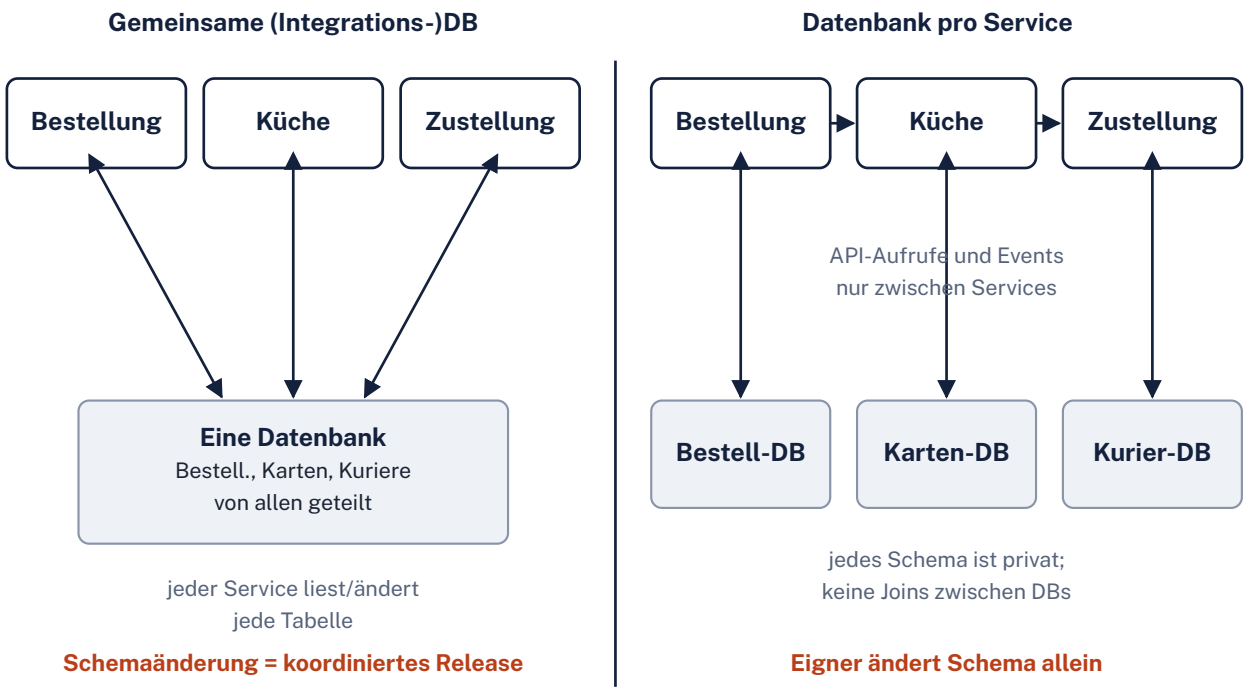
Die gemeinsame Datenbank und warum sie Services koppelt

Der einfachste Weg, mehrere Anwendungen mit denselben Daten arbeiten zu lassen, ist, ihnen allen Zugriff auf eine Datenbank zu geben. Martin Fowler nannte das eine Integrationsdatenbank (2004): eine Datenbank, die der Integrationspunkt zwischen Anwendungen ist, im Gegensatz zu einer *Anwendungsdatenbank*, die nur einer Anwendung dient. Sie wirkt verlockend: keine APIs zu bauen, Joins funktionieren über alles, und eine Transaktion kann Bestellungen und Zahlungen gemeinsam aktualisieren. Fowlers Urteil, das die meisten Architekten teilen, lautet, dass Integrationsdatenbanken vermieden werden sollten, weil die Kopplung, die sie erzeugen, tief und größtenteils unsichtbar ist.

Betrachten Sie die Formen von Kopplung, die sie mit sich bringt. **Schemakopplung**: Jede Tabelle und jede Spalte ist faktisch eine öffentliche API. Wer `orders.state` in `orders.status` umbenennt, bricht jeden Service, der sie liest, und niemand kann sicher sagen, welche das sind, weil SQL-Abfragen über viele Codebasen, Berichte und Tabellenkalkulationen verstreut sind. **Deployment-Kopplung**: Eine Schemaänderung muss mit jedem Team abgestimmt werden, das die Tabelle berührt, sodass Services, die unabhängig releast werden sollten, gemeinsam releast werden. **Semantische Kopplung**: Das Schema muss jedem Client dienen, also wird es allgemeiner und verwirrender (Fowler merkt an, dass es am Ende Konzepte zu vereinheitlichen versucht, die in getrennte Bounded Contexts gehören), und die Geschäftsregeln für eine Tabelle landen dupliziert, und zwar uneinheitlich, in jedem Service, der sie schreibt. **Laufzeitkopplung**: Die langsame Berichtsabfrage oder der sperrintensive Batch-Job eines Service beeinträchtigt jeden anderen Service, und die Datenbank wird zu einem gemeinsamen Single Point of Failure und zu einer Obergrenze für Verbindungen.

Nichts davon ist fatal für eine einzelne Anwendung oder einen modularen Monolithen, dessen Module die Tabellen der anderen respektieren, oder für die grobgranulare servicebasierte Architektur, in der sich eine Handvoll Domänenservices oft bewusst eine Datenbank teilt. Zum Problem wird es, wenn Teams unabhängige Deploybarkeit erwarten, aber die gemeinsamen Tabellen behalten; das ist ein klassischer Weg zu einem *verteilten Monolithen*.

Gemeinsame Datenbank im Vergleich zu Datenbank pro Service



Bei einer gemeinsamen Datenbank ist jede Tabelle eine öffentliche Schnittstelle; bei Datenbank pro Service ist jedes Schema privat, und Daten überqueren Grenzen nur über APIs und Events.

Datenbank pro Service

Die Alternative, die mit Microservices verbunden ist, ist das Muster Datenbank pro Service: Die persistenten Daten jedes Service sind privat, und andere Services erreichen sie nur über die API des Service oder die Events, die er veröffentlicht. Chris Richardsons Musterkatalog (microservices.io) merkt an, dass das nicht einen eigenen Server pro Service bedeuten muss. Es gibt drei Stufen der Trennung, und die wichtige Grenze ist die erste.

Drei Wege, die Daten eines Service privat zu halten

Niveau	Was getrennt ist	Stärke der Grenze	Typische Verwendung
Eigene Tabellen pro Service	Jeder Service hat seine eigenen Tabellen in einer gemeinsamen Datenbank, mit eigenen Zugangsdaten, die nur diese Tabellen erreichen.	Am schwächsten: beruht auf Berechtigungen und Disziplin; teilt weiterhin Kapazität und Ausfälle.	Früh in einer Migration; kleine Systeme; modulare Monolithen.
Schema pro Service	Jeder Service hat sein eigenes Schema (Namensraum) auf einem gemeinsamen Server.	Klare Datenhoheit und Zugriffskontrolle; teilt weiterhin Performance und Verfügbarkeit des Servers.	Die meisten Teams, die mit mehreren Services beginnen; günstiger im Betrieb.
Datenbankserver pro Service	Jeder Service hat seine eigene Datenbankinstanz, möglicherweise eines anderen Typs.	Am stärksten: getrennte Skalierung, getrennte Ausfälle und freie Technologiewahl.	Services mit sehr unterschiedlicher Last, Datenform oder unterschiedlichem Verfügbarkeitsbedarf.

Die Vorteile spiegeln die Kosten der gemeinsamen Datenbank: Eigentümer ändern ihre Schemas frei, skalieren und optimieren ihre Speicher unabhängig, wählen die richtige Art von Datenbank und begrenzen Ausfälle. Die Kosten sind real und sollten in jedem Entscheidungsdokument genannt werden. Joins über Services hinweg entfallen und müssen durch API-Komposition oder replizierte Daten ersetzt werden. Eine einzelne ACID-Transaktion kann nicht mehr Bestellungen und Zahlungen zugleich umfassen; serviceübergreifende Änderungen werden zu Sagas mit kompensierenden Schritten. Referenzielle Integrität über Services hinweg (jede Bestellung verweist auf einen echten Kunden) wird nicht mehr durch Fremdschlüssel durchgesetzt. Reporting braucht eine eigene Pipeline. Und es gibt mehr Datenbanken, die gesichert, gepatcht und überwacht werden müssen.

The Hard Parts fasst die Entscheidung als Gleichgewicht von Kräften. **Disintegratoren** drängen zum Aufteilen der Daten: Änderungskontrolle (wie viele Services brechen, wenn sich eine Tabelle ändert), Verbindungslimits, Skalierbarkeit, Fehlertoleranz, der Wunsch nach unabhängig deploybaren Einheiten und der Bedarf an unterschiedlichen Datenbanktypen. **Integratoren** ziehen Daten zusammen: Beziehungen, die referenzielle Integrität brauchen, und Operationen, die wirklich eine einzige atomare Transaktion brauchen. Wenn zwei Tabellen immer gemeinsam in einer Transaktion geändert werden und kein unabhängiges Eigenleben haben, spricht das stark dafür, dass sie demselben Eigentümer gehören.

Wenn mehrere Services dieselben Daten schreiben

Fragen der Datenhoheit werden schwierig, wenn mehr als ein Service dieselbe Entität berechtigterweise ändert. *The Hard Parts* unterscheidet drei Situationen. Bei **alleiniger Datenhoheit** schreibt nur ein Service die Tabelle, und er besitzt sie einfach. Bei **allgemeiner Datenhoheit** schreiben die meisten Services dieselbe Tabelle (etwa ein Audit-Log oder eine Reihe von Konfigurationseinstellungen); die übliche Antwort ist, einen eigenen Service zu schaffen, der die Daten besitzt, und allen anderen eine API zu geben, oft eine asynchrone, damit die Schreiber nicht ausgebremst werden. Bei gemeinsamer Datenhoheit schreibt eine kleine Zahl von Services, typischerweise zwei, dieselbe Tabelle, meist innerhalb derselben Domäne. Das Buch bietet vier Techniken für gemeinsame Datenhoheit an.

Gemeinsame Datenhoheit auflösen

Technik	So funktioniert es	Wichtigste Abwägung
Tabelle aufteilen	Die Tabelle so aufteilen, dass jeder Service die Spalten, die er schreibt, in einer eigenen Tabelle besitzt. Aus der Zeile in `orders` der Essenslieferplattform werden eine Bestellung im Besitz von Bestellungen und ein Küchenbon im Besitz der Küche.	Klare Datenhoheit, aber das Anlegen oder Löschen der übergeordneten Entität muss nun zwischen Services synchronisiert werden, und jede Gesamtsicht braucht beide.
Datendomäne	Die Tabelle für die wenigen beteiligten Services gemeinsam behalten, in einem kleinen gemeinsamen Schema, das sie gemeinsam kontrollieren.	Einfach und schnell, Transaktionen bleiben möglich, aber es ist eine bewusst begrenzte gemeinsame Datenbank: Die Services sind bei Schemaänderungen gekoppelt, und die Grenze zwischen ihnen wird schwächer.
Delegieren	Einen Eigentümer wählen; der andere Service schickt ihm Anfragen (synchrone Aufrufe oder asynchrone Befehle), statt selbst zu schreiben. Der Eigentümer wird nach der primären Domäne gewählt (wessen Konzept ist es?) oder nach betrieblichen Eigenschaften (welcher Service braucht die schnellsten, am besten verfügbaren Schreibvorgänge?).	Klare Datenhoheit und Regeln an einem Ort, um den Preis eines zusätzlichen Netzwerk-Hops, einer neuen Ausfallart und möglicherweise Eventual Consistency für den delegierenden Service.
Services zusammenlegen	Die Services, die sich die Daten teilen, zu einem Service zusammenführen.	Beseitigt das Problem und stellt lokale Transaktionen wieder her, schafft aber einen größeren Service mit gröberem Deployment, gröberer Skalierung und gröberer Fehlerisolation.

Keine Technik ist kostenlos, und die richtige hängt davon ab, wie oft sich die Daten ändern, wie streng ihre Regeln sind und wie unabhängig die Teams sein müssen. In der Praxis ist Delegation die häufigste Antwort für Geschäftsentitäten mit einem Lebenszyklus (Bestellungen, Zahlungen, Buchungen), das Aufteilen der Tabelle funktioniert gut, wenn die gemeinsam geschriebenen Spalten eigentlich verschiedene Tatsachen sind, die nur zufällig zusammen gespeichert wurden, und das Zusammenlegen ist die ehrliche Antwort, wenn die beiden Services von vornherein zu fein geschnitten wurden.

Daten lesen, die Ihnen nicht gehören

Sobald die Datenhoheit geklärt ist, muss jeder andere Service die Daten immer noch *lesen*. Die Küchenanzeige braucht die Lieferhinweise des Kunden; die Betrugserkennung braucht die jüngsten Zahlungen des Kontos. Es gibt vier grundlegende Optionen, und jede wägt Aktualität gegen Kopplung ab. Beachten Sie, dass ereignisgesteuerte Designs, die in Kapitel 6 behandelt wurden, die dritte und vierte liefern.

Wege, die Daten eines anderen Service zu lesen

Option	So funktioniert es	Aktualität	Kopplung und Kosten
Synchroner Aufruf	Die API des Eigentümers in dem Moment fragen, in dem die Daten gebraucht werden.	Aktuell, zum Zeitpunkt des Aufrufs.	Laufzeitkopplung: Ist der Eigentümer langsam oder ausgefallen, gilt das auch für den Aufrufer; Latenzen summieren sich entlang einer Kette.
Column Schema Replication	Die wenigen benötigten Spalten (zum Beispiel Name und Adresse des Restaurants) in die eigene Datenbank des Lesers kopieren, aktuell gehalten durch Events oder Change Data Capture.	Eventually consistent: veraltet um die Replikationsverzögerung.	Schnelle lokale Lesezugriffe und Joins; der Leser muss mit verspäteten oder verpassten Aktualisierungen umgehen und darf die Kopie nie beschreiben.
Replizierter Cache	Ein In-Memory Data Grid (etwa Hazelcast oder Apache Ignite) hält in jedem konsumierenden Service eine Kopie der Daten des Eigentümers, synchronisiert durch das Grid.	Nahezu in Echtzeit.	Sehr schnelle Lesezugriffe; Speicherkosten, Laden beim Start und die betriebliche Komplexität des Grids.
Events (Event-Carried State Transfer)	Der Eigentümer veröffentlicht Events, die die geänderten Daten enthalten; Leser behalten, was sie brauchen.	Eventually consistent.	Lose zeitliche Kopplung; Leser müssen mit Duplikaten, Reihenfolge und Schemaevolution umgehen.

Fowler nennt die vierte Option Event-Carried State Transfer: Events wie `MenuItemPriceChanged` enthalten die neuen Werte, sodass Konsumenten nicht zurückrufen müssen. Ein verwandter Mechanismus, Change Data Capture (CDC), liest das Log der Datenbank des Eigentümers und macht aus jeder committeten Änderung ein Event; das vermeidet das *Dual-Write-Problem* (in die Datenbank schreiben und getrennt davon ein Event veröffentlichen, sodass das eine gelingen und das andere scheitern kann). Die Transactional Outbox, die dasselbe Problem innerhalb der Anwendung löst, wird bei den Sagas behandelt.

ACHTUNG

Eine Kopie ist ein Cache, kein zweiter Master

Replizierte Daten sind nur sicher, wenn sie als schreibgeschützt und möglicherweise veraltet behandelt werden. Sobald ein Konsument anfängt, seine Kopie zu korrigieren (die Adresse eines Restaurants lokal zu berichtigen, weil die Aktualisierung des Eigentümers verspätet war), gibt es zwei Systems of Record, und sie werden auseinanderlaufen. Legen Sie für jede Kopie fest, wie veraltet sie sein darf, was geschieht, wenn sie falsch ist, und wie sie vom Eigentümer neu aufgebaut wird, wenn sie verloren geht.

WENN DAS NEU FÜR SIE IST

Schritt für Schritt: den Eigentümer fragen oder eine Kopie halten

Der schwierigste Gedanke dieses Abschnitts ist, wie ein Service Daten nutzen sollte, die ihm nicht gehören. Er kann jedes Mal den Eigentümer fragen oder eine eigene Kopie halten, und jede Wahl wägt Aktualität gegen Abhängigkeit vom Eigentümer ab.

- Um Kuriere an die richtige Tür zu schicken, muss die Lieferung wissen, wo jedes Restaurant liegt und wie es heißt. Diese Daten gehören dem Restaurant-Service, der ihr System of Record ist.
- Option eins: Die Lieferung ruft jedes Mal, wenn sie einen Kurier zuweist, die API des Restaurant-Service auf. Die Antwort ist immer aktuell, aber wenn der Restaurant-Service langsam oder ausgefallen ist, werden auch die Kurierzuweisungen langsamer oder stocken, obwohl mit der Lieferung selbst alles in Ordnung ist.
- Option zwei: Die Lieferung kopiert genau diese beiden Spalten in ihre eigene Datenbank; die Tabelle oben nennt das Column Schema Replication. Immer wenn sich die Angaben eines Restaurants ändern, veröffentlicht der Restaurant-Service ein Event mit den neuen Werten, und die Lieferung aktualisiert ihre Kopie: der Event-Carried State Transfer aus Kapitel 6. Lesezugriffe sind jetzt lokal und schnell und funktionieren weiter, während der Restaurant-Service ausgefallen ist.
- Der Preis ist Veralterung. Wenn ein Restaurant seine Adresse korrigiert, hält die Lieferung die alte, bis das Event ankommt. Geht das Event verloren, bleibt die Kopie falsch, bis sie vom Eigentümer aufgefrischt wird.
- Deshalb ist die Kopie schreibgeschützt. Bemerkt die Lieferung eine falsche Adresse, teilt sie das dem Restaurant-Service mit, der sie korrigiert und die Berichtigung veröffentlicht; die Kopie zu bearbeiten würde einen zweiten Master schaffen, wie die Warnung oben erklärt.
- **Die Lehre:** Den Eigentümer zu fragen liefert frische Daten, bindet Sie aber zur Laufzeit an ihn; eine Kopie macht Sie unabhängig, ist aber schreibgeschützt und möglicherweise veraltet. Legen Sie für jede Kopie fest, wie veraltet sie sein darf und wie sie vom Eigentümer neu aufgebaut wird.

Reporting über Services hinweg

Datenbank pro Service erschwert eine alte Aufgabe: den übergreifenden Bericht. „Umsatz pro Restaurant pro Woche, abzüglich Erstattungen, aufgeschlüsselt nach Kuriertyp“ braucht Bestellungen, Zahlungen, Restaurants und Lieferungen. Ihn durch seitenweises Aufrufen von vier APIs zu erstellen ist langsam und belastet operative Services mit Reporting-Last. Ihre Datenbanken direkt abzufragen führt die Kopplung der gemeinsamen Datenbank durch die Hintertür wieder ein.

Die übliche Antwort ist, Reporting vom operativen Betrieb zu trennen. Fowler beschrieb eine *Reporting-Datenbank*: einen eigenen Speicher, für Abfragen gestaltet und aus den operativen Systemen gespeist, den Reporting-Nutzer frei lesen dürfen. Heute ist das meist ein Data Warehouse oder Lakehouse, gespeist durch CDC oder Events, behandelt in Analytics und Streaming. Für kleine, nahezu echtzeitnahe Ansichten innerhalb des Produkts (das Dashboard eines Restaurants mit den heutigen Bestellungen) kann ein Service aus Events sein eigenes Read Model aufbauen, die Idee hinter CQRS. In beiden Fällen hängen die Pipelines, die das Reporting speisen, von den Datenstrukturen der Eigentümer ab, und deshalb brauchen sie Verträge.

Datenverträge

Ein Datenvertrag ist eine explizite, versionierte Vereinbarung zwischen dem Eigentümer eines Datenbestands und seinen Konsumenten. Er legt das Schema fest (Felder, Typen, welche null sein dürfen), die Bedeutung jedes Felds (ist `amount` in Pence oder Pfund, mit oder ohne Mehrwertsteuer?), Qualitätserwartungen (keine doppelten Bestellkennungen, höchstens 0,1 % der Zeilen ohne Restaurant), Service-Levels (wie aktuell die Daten sind, wie lange sie aufbewahrt werden), wer sie verantwortet und wie Änderungen angekündigt werden. Er überträgt die Idee eines API-Vertrags auf Daten, die über Events, CDC-Streams und Warehouse-Tabellen zwischen Teams fließen.

Verträge sind am nützlichsten, wenn sie maschinenlesbar sind und automatisch geprüft werden: eine Pipeline, die eine Schemaänderung ablehnt, die registrierte Konsumenten brechen würde, oder ein Test, der fehlschlägt, wenn eine Qualitätsregel verletzt wird. Der Open Data Contract Standard (ODCS), gepflegt vom Bitol-Projekt unter der Linux Foundation AI & Data, definiert ein YAML-Format für Schema, Qualitätsregeln, Service-Levels, Rollen und Support-Kanäle. Datenverträge sind nicht kostenlos: Jemand muss sie schreiben und pflegen, und zu strenge Verträge bremsen Änderungen. *The Hard Parts* stellt strenge Verträge (präzise, überprüfbar, eng koppelnd) lockeren gegenüber (leichter weiterzuentwickeln, aber mit mehr Testbedarf) und schlägt Consumer-Driven Contracts, bei denen Konsumenten angeben, was sie tatsächlich nutzen, als Mittelweg vor. Wie man ein Schema ändert, ohne Konsumenten zu brechen, ist Thema der Schemaevolution.

AUSGEARBEITETES BEISPIEL

Ausgearbeitetes Beispiel: Wem gehört der Bestellstatus in der Essenslieferplattform?

Die Plattform hat drei Services, die sich für den Fortschritt einer Bestellung interessieren: Bestellungen (nimmt Bestellungen an und kassiert), Restaurant (das Küchentablet: nimmt Bestellungen an, meldet das Essen als fertig) und Lieferung (weist Kuriere zu, verfolgt Abholung und Zustellung). Heute schreiben alle drei die Spalte `status` einer gemeinsamen Tabelle `orders`. Kunden haben gesehen, wie Bestellungen von „Abgeholt“ zurück auf „In Zubereitung“ springen, und niemand kann sagen, wo die Stornierungsregeln liegen.

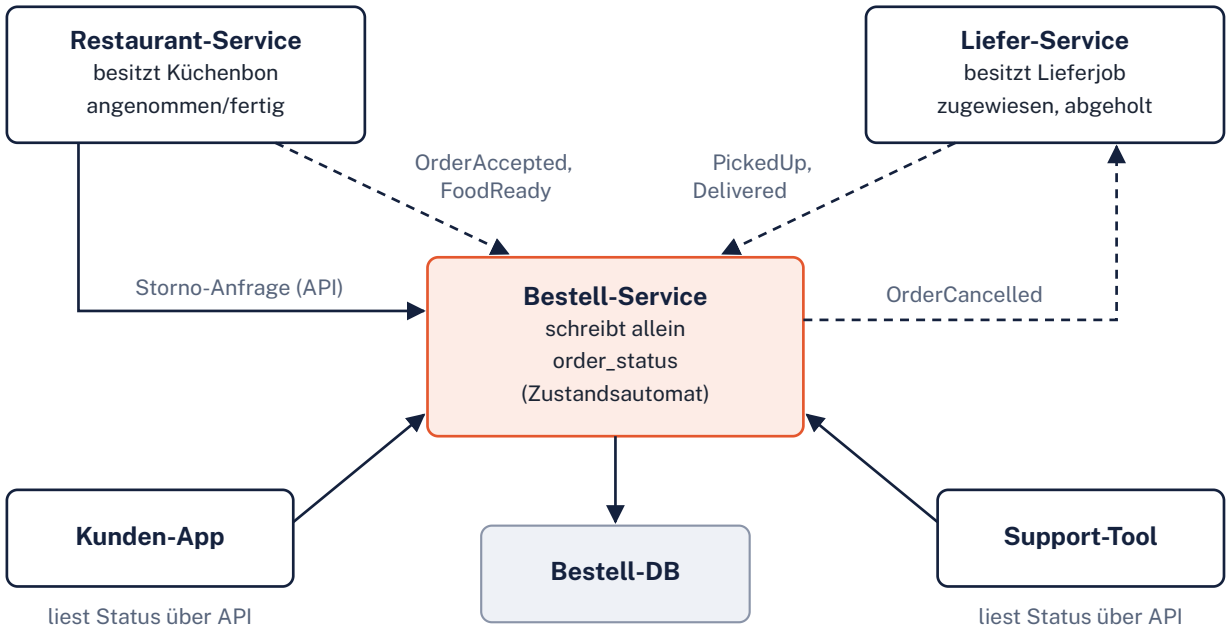
Diagnose	Gemeinsame Datenhoheit über eine Spalte mit drei Schreibern. Jeder schreibt, was er weiß, der letzte Schreibvorgang gewinnt, und verspätete Nachrichten setzen den Status zurück. Die Regel „eine Bestellung kann nur storniert werden, bevor die Küche beginnt“ wird in zwei Services geprüft, und zwar unterschiedlich.
Die Tatsachen trennen	Eigentlich gibt es drei Tatsachen: den Zustand des Küchenbons (im Besitz des Restaurants), den Zustand des Lieferauftrags (im Besitz der Lieferung) und den Bestellstatus, den der Kunde sieht, der beide kombiniert und die Geschäftsregeln trägt (Stornierung, Erstattungen, Entschädigung).
Option: Tabelle aufteilen	Jedem Service eine eigene Statusspalte geben und die Leser sie kombinieren lassen. Verworfen: Jeder Leser (App, Support-Tool, Benachrichtigungen) müsste die Kombination neu implementieren, und die Stornierungsregel hätte immer noch kein Zuhause.
Option: Services zusammenlegen	Alle drei zu einem Service zusammenführen. Verworfen: drei Teams, sehr unterschiedliche Last (Standortaktualisierungen der Kuriere kommen alle paar Sekunden) und unterschiedliche Release-Zyklen.
Entscheidung: delegieren	Bestellungen ist der einzige Schreiber von `order_status`. Restaurant und Lieferung besitzen ihre eigenen Tatsachen und veröffentlichen Events (`OrderAccepted`, `FoodReady`, `PickedUp`, `Delivered`). Bestellungen konsumiert sie und wendet einen Zustandsautomaten an. Eine Stornierung vom Restaurant-Tablet ist eine Anfrage an die API von Bestellungen, die sie annimmt oder ablehnt.
Zustandsautomat	PLACED, ACCEPTED, PREPARING, READY, PICKED_UP, DELIVERED, wobei CANCELLED nur aus PLACED oder ACCEPTED erlaubt ist. Übergänge gehen nur vorwärts, sodass ein verspätetes `FoodReady`, das nach `PickedUp` ankommt, aufgezeichnet wird, den Status aber nicht ändert; Duplikate werden ignoriert.

Konsequenzen	Ein Ort für die Regeln; keine Rücksprünge; jedes Team releast unabhängig. Kosten: Der Status hinkt der Küche um die Event-Verzögerung hinterher (typischerweise deutlich unter einer Sekunde), Bestellungen muss verfügbar sein, um Stornierungen anzunehmen, und die Events werden zu einem Vertrag, der versioniert werden muss.

Der entscheidende Schritt war nicht, eine Technik aus der Tabelle zu wählen, sondern zu bemerken, dass eine Spalte drei verschiedene Tatsachen mit drei natürlichen Eigentümern verbarg.

Halten Sie die Entscheidung als Architecture Decision Record fest, einschließlich der verworfenen Optionen; das nächste Team, das fragt „Warum können wir den Status nicht einfach aus der Kurier-App aktualisieren?“, wird ihn brauchen.

Bestellstatus nach der Entscheidung



Jede Tatsache hat genau einen Schreiber; alle anderen lesen/fragen

Jede Tatsache hat einen Schreiber: Restaurant und Lieferung veröffentlichen, was ihnen gehört, und allein der Bestell-Service macht aus diesen Events den Status, den der Kunde sieht.

Häufige Fehler

Services aufteilen, aber die gemeinsamen Tabellen behalten. Teams bekommen die Betriebskosten der Verteilung ohne die Unabhängigkeit, die sie erkaufen sollte. Wenn sich Services nicht ohne eine abgestimmte Schemaänderung deployen lassen, sind sie nicht unabhängig. **Entity-Services.** Ein `CustomerService`, der nichts als Anlegen, Lesen,

Aktualisieren und Löschen über einer Tabelle bietet und von jedem anderen Service aufgerufen wird, ist eine gemeinsame Datenbank mit zusätzlicher Latenz; Datenhoheit sollte Geschäftsfähigkeiten und ihren Regeln folgen, nicht Substantiven. **Kopien, die unbemerkt zu Mastern werden.** Siehe die Warnung oben. **Reporting bis zum Schluss ignorieren.** Der erste Finanzbericht nach der Aufteilung zeigt oft, dass niemand geplant hat, wie die Daten wieder zusammengeführt werden. **Datenhoheit als rein technisch behandeln.** Datenhoheit ist auch organisatorisch: Jemand muss für die Qualität der Daten geradestehen, und nach dem Gesetz von Conway halten Datengrenzen, die quer zu Teamgrenzen verlaufen, meist nicht lange.

Ein guter Test für jedes Datendesign besteht darin, für jede wichtige Tatsache drei Fragen zu stellen: Wer ist der einzige Schreiber, wo werden ihre Regeln durchgesetzt, und wie veraltet darf jede Kopie sein? Lautet eine Antwort „kommt darauf an“ oder „mehrere Services“, ist das Design nicht fertig.

PRÜFEN SIE IHR VERSTÄNDNIS

▼ **Zwei Services aktualisieren jeweils die Spalte `status` derselben Tabelle. Um welche Art von Datenhoheit handelt es sich, und nennen Sie zwei Wege, sie aufzulösen.**

Gemeinsame Datenhoheit. Sie lässt sich auflösen, indem man die Tabelle aufteilt, Schreibvorgänge an einen Eigentümer delegiert, eine kleine gemeinsame Datendomäne behält oder die Services zusammenlegt.

▼ **Warum untergräbt eine gemeinsame Datenbank unabhängiges Deployment?**

Jede Tabelle ist faktisch eine öffentliche Schnittstelle, sodass eine Schemaänderung mit jedem Service abgestimmt werden muss, der sie liest oder schreibt, was gemeinsame Releases erzwingt.

▼ **Was ist beim Lesen der Daten eines anderen Service der Unterschied zwischen Column Schema Replication und einem synchronen Aufruf?**

Replikation hält eine lokale, eventually consistent Kopie, die schnell und verfügbar ist, auch wenn der Eigentümer ausgefallen ist; ein synchroner Aufruf liefert aktuelle Daten, koppelt aber Verfügbarkeit und Latenz des Aufrufers an den Eigentümer.

▼ **Was fügt ein Datenvertrag über ein Schema hinaus hinzu?**

Die Bedeutung der Felder, Qualitätserwartungen, Vorgaben zu Aktualität und Aufbewahrung, die Verantwortung sowie die Art, wie Änderungen angekündigt und versioniert werden.

▼ **Probieren Sie es aus: Booking, Payments und die Box Office aktualisieren alle die Spalte status einer gemeinsamen Tabelle tickets, und Tickets springen manchmal von Refunded zurück auf Paid. Wer sollte Eigentümer von status sein, und wie sollten die anderen den Wert ändern oder lesen?**

Benennen Sie einen Schreiber. Booking sollte den Ticketstatus besitzen und die erlaubten Änderungen an einer Stelle durchsetzen, einschließlich der Regel, dass Refunded nicht zu Paid werden kann. Payments und das Box Office schreiben ihn nicht; sie senden ein Kommando oder veröffentlichen ein Ereignis wie RefundCompleted, und Booking entscheidet. Andere lesen über die API von Booking oder halten schreibgeschützte Kopien und nehmen in Kauf, dass diese etwas veraltet sein können. Die eigenen Details jedes Teams, etwa der Zahlungsstatus, bleiben in dessen eigenem Speicher. Der Abschied von der gemeinsamen Tabelle kommt zuletzt.

QUELLEN

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

Die Wahl einer Datenbank: relational, Dokument, Key-Value, Spalten, Graph und Zeitreihen

BEVOR SIE DAS LESEN

Dieser Abschnitt baut auf:

- 0.5 Daten speichern
- 6.6 CAP und Konsistenz
- 7.1 Datenhoheit

Begriffe von früher, die er verwendet: System of Record, Transaktion, SQL (Structured Query Language), Change Data Capture (CDC), Architecture Decision Record (ADR)

DIE IDEE IN KÜRZE

Eine Datenbank zu wählen ist wie die Wahl von Stauraum für eine Wohnung. Ein Aktenschrank mit beschrifteten Mappen (eine relationale Datenbank) eignet sich für Unterlagen, die Sie miteinander abgleichen müssen. Ein Kleiderschrank mit kompletten Outfits (ein Dokumentenspeicher) eignet sich für Dinge, die Sie immer zusammen herausnehmen. Eine Reihe nummerierter Schließfächer (ein Key-Value-Store) ist am schnellsten, wenn Sie die Nummer kennen. Eine Pinnwand mit Fäden zwischen den Nadeln (ein Graph) zeigt, wer wen kennt. Die meisten Wohnungen brauchen einen Aktenschrank plus ein oder zwei Spezialmöbel, nicht von allem eines, denn jedes zusätzliche Stück braucht Platz, Pflege und jemanden, der weiß, wie es funktioniert.

WAS SIE WISSEN MÜSSEN

- Relationale Datenbanken bleiben der Standard für Geschäftsdaten, weil sie Transaktionen, Constraints, flexible Abfragen, ausgereifte Werkzeuge und weit verbreitete Kenntnisse bieten und inzwischen auch JSON-, Text-, Geo- und Vektordaten in moderatem Umfang verarbeiten.
- Dokument-, Key-Value-, Wide-Column-, Graph-, Zeitreihen-, Such- und Vektorspeicher machen jeweils bestimmte Zugriffsmuster günstig und andere teuer; ihre Stärke liegt als spezialisierte oder abgeleitete Speicher.
- Distributed SQL (Spanner, CockroachDB, YugabyteDB, TiDB) behält SQL und Transaktionen bei und skaliert über Knoten und Regionen, um den Preis von Schreiblatenz, Geld und betrieblicher Komplexität.

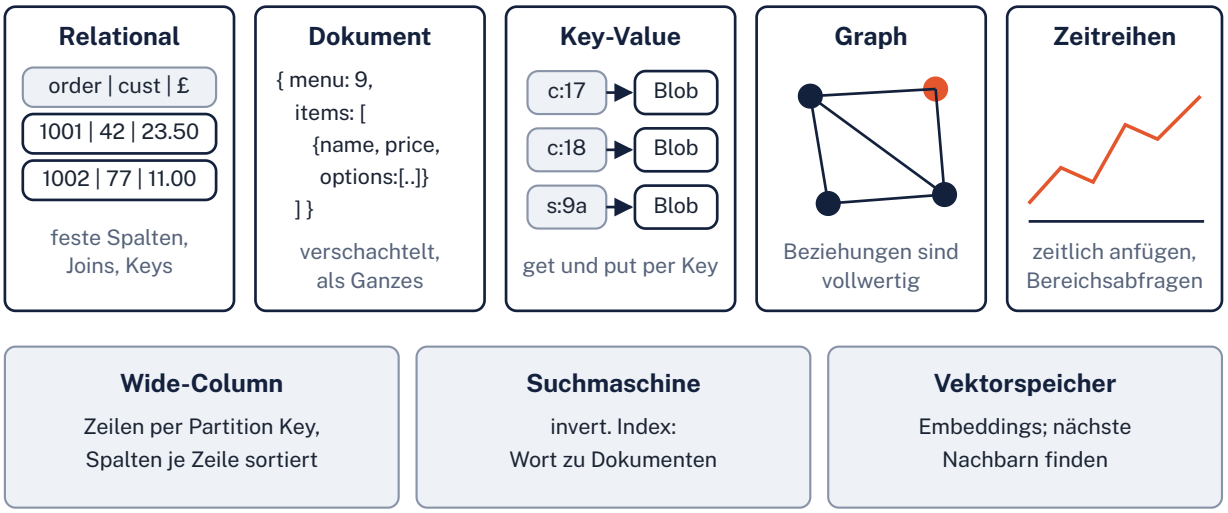
- Wählen Sie pro Datenbestand anhand von Belegen: Datenform, Zugriffsmuster, Konsistenzbedarf, gemessene Größenordnung, Betriebsaufwand und Teamkenntnisse; schreiben Sie die wichtigsten Abfragen und Volumina zuerst auf.
- Trennen Sie Systems of Record, die die konservativste Technologie brauchen, von abgeleiteten Speichern wie Suchindizes und Caches, die spezialisiert sein können, weil sie sich neu aufbauen lassen.
- Polyglotte Persistenz erlaubt jedem Datenbestand einen passenden Speicher, aber jede zusätzliche Technologie bringt dauerhafte betriebliche und kognitive Kosten mit sich; McKinleys „Choose Boring Technology“ (2015) plädiert dafür, Innovation sparsam einzusetzen.

Warum die Wahl Sorgfalt verdient

Eine Datenbank gehört zu den am schwersten austauschbaren Komponenten. Anwendungen werden um ihre Daten herum neu geschrieben, nicht umgekehrt; Terabytes an Live-Daten zu migrieren, während das Geschäft weiterläuft, kann ein Jahr dauern. Die Wahl des Speichers ist deshalb eine Einwegtür, die eine explizite Entscheidung verdient, während viele Teams sie aus Gewohnheit oder nach Mode treffen.

Kapitel 0 hat relationale Datenbanken und die vier NoSQL-Familien eingeführt. Dieser Abschnitt geht weiter: wofür jede Familie wirklich gut ist, die neueren Kategorien (Zeitreihen, Suche, Vektoren und Distributed SQL), wie man nach Belegen statt nach Begeisterung wählt und was es kostet, mehrere Arten zugleich zu betreiben. Die Antwort ist selten eine Datenbank für alles und fast nie eine andere Datenbank für jeden Service.

Datenmodelle auf einen Blick



Jede Familie organisiert Daten anders, und diese Form entscheidet, welche Abfragen günstig und welche unmöglich sind.

Relational: weiterhin der Standard, aus guten Gründen

Eine relationale Datenbank speichert Daten in Tabellen mit deklarierten Typen, Schlüsseln und Constraints und beantwortet deklarative SQL-Abfragen, eine Idee aus Edgar Codds Paper von 1970. Fünfzig Jahre später ist sie aus vier Gründen weiterhin der vernünftige Standard für Geschäftsdaten. Erstens **Transaktionen und Constraints**: Die Datenbank selbst garantiert, dass eine Zahlung und ihre Hauptbucheinträge gemeinsam committet werden und dass keine Bestellung auf einen fehlenden Kunden verweist, sodass diese Regeln nicht in jedem Service unvollkommen neu implementiert werden müssen. Zweitens **Flexibilität bei Abfragen**: Sie müssen beim Entwurf des Schemas nicht jede künftige Frage kennen; ein neuer Bericht ist eine neue Abfrage, keine neue Tabelle. Drittens **Reife**: Jahrzehnte an Werkzeugen für Backup, Replikation, Monitoring, Migration und Performance-Diagnose sowie Fachleute, die sie schon kennen. Viertens **Bandbreite**: Moderne relationale Systeme speichern und indizieren JSON (`jsonb` in PostgreSQL), unterstützen Volltextsuche, Geodaten (PostGIS), Zeitpartitionierung und Vektorähnlichkeit (die Erweiterung pgvector), sodass eine gut betriebene Datenbank in moderatem Umfang viele Anforderungen abdecken kann.

Die echten Grenzen sind konkret. Schreibvorgänge auf einem einzelnen Primary sind durch eine Maschine begrenzt; sehr große Tabellen brauchen sorgfältige Partitionierung; Schemaänderungen an riesigen Tabellen brauchen Technik (Schemaevolution); und manche Zugriffsmuster, etwa tiefe Graphtraversierungen oder relevanzbasierte Textsuche, bedienen

spezialisierte Engines deutlich besser. Ein einzelner moderner Server mit schnellen SSDs kann Tausende Transaktionen pro Sekunde und Terabytes an Daten bewältigen, was den Bedarf der meisten Organisationen abdeckt.

Identifikatoren wählen: Sequenzen, UUIDv7 und ULID

Schlüssel lassen sich später nur schwer ändern. Eine **Sequenz** ist kompakt und geordnet, braucht aber einen einzigen Vergeber, und die Bestellnummer 1042 verrät Außenstehenden Ihr Volumen. Eine zufällige **UUIDv4** lässt sich überall erzeugen, doch RFC 9562 warnt vor schlechter Indexlokalität: Einfügungen landen an zufälligen Stellen eines B-Baums, und der Effekt „kann dramatisch sein“. **UUIDv7** stellt einen 48-Bit-Millisekunden-Zeitstempel voran und lässt Zufallsbits folgen, sodass neue Werte nah beieinanderliegen; PostgreSQL 18 enthält `uuidv7()`, und der RFC rät, UUIDs binär zu speichern. **ULID** ist ähnlich: 48 Bit Zeit, 80 Bit Zufall und 26 Textzeichen. Zeitlich geordnete IDs verraten, wann eine Zeile angelegt wurde, und können bei Key-Range-Partitionierung einen Hot Range erzeugen. Der RFC warnt außerdem davor anzunehmen, UUIDs seien schwer zu erraten; die Autorisierung darf sich also nicht auf sie stützen.

Die nicht relationalen Familien

Eine Dokumentendatenbank speichert jeden Datensatz als in sich geschlossenes Dokument, meist JSON oder eine binäre Form davon. Sie passt zu *Aggregaten* im Sinne des Domain-Driven Design: Dingen, die als Ganzes gelesen und geschrieben werden, etwa die Speisekarte eines Restaurants mit ihren Rubriken, Gerichten und Optionsgruppen. Ein Dokument zu lesen ist ein einziger Zugriff ohne Joins. Die Abwägung zeigt sich, wenn dieselben Daten in einer anderen Form abgefragt (alle Gerichte mit Erdnüssen über alle Speisekarten hinweg) oder in vielen Dokumenten zugleich geändert werden müssen; Joins und dokumentübergreifende Transaktionen gibt es in modernen Produkten, aber sie sind teurer und weniger natürlich als in SQL.

Ein Key-Value-Store ist ein verteiltes Wörterbuch: einen Wert unter einem Schlüssel ablegen, ihn wieder abrufen. Weil der Speicher nie in die Werte hineinschaut, kann er extrem schnell und leicht zu partitionieren sein. Redis (seit 2024 unter Source-Available-Lizenzen, seit Redis 8 mit einer Open-Source-Option unter AGPL) und Valkey, sein BSD-lizenzierter Fork, halten Daten im Arbeitsspeicher mit optionaler Persistenz und bieten zusätzlich Datenstrukturen (Sorted Sets, Streams, Geo-Indizes) und Ablaufzeiten; Amazon DynamoDB ist ein verwalteter, partitionierter Key-Value- und Dokumentenspeicher. Sie eignen sich für Sessions, Warenkörbe, Zähler für Rate Limiting, Feature-Flags, Caches und die letzte Position jedes Kuriers.

Ein Wide-Column-Store (Bigtable, Apache Cassandra, ScyllaDB, HBase) gruppiert Zeilen nach einem Partition Key und speichert sie innerhalb jeder Partition nach einem Clustering Key sortiert. Er ist für massive Schreibvolumina ausgelegt, verteilt über viele Knoten und Rechenzentren. Die Disziplin, die er verlangt, ist *Query-First-Modellierung*: Sie entwerfen eine Tabelle pro Abfrage und duplizieren Daten großzügig, weil Abfragen, die nicht beim Partition Key ansetzen, teuer oder unmöglich sind. Cassandra stammt zum Teil von Amazons Dynamo-Paper von 2007 ab, dessen Replikationsideen im übernächsten Abschnitt vorkommen.

Eine Graphdatenbank speichert Knoten und Beziehungen als vollwertige Datensätze und beantwortet Traversierungsabfragen wie „Kunden innerhalb von drei Schritten von diesem Betrugsring“ ohne die kaskadierenden Joins, die solche Abfragen in SQL langsam machen. Cypher von Neo4j ist die bekannteste Abfragesprache, und die ISO hat 2024 eine standardisierte Graphabfragesprache veröffentlicht, GQL (ISO/IEC 39075). Graphen eignen sich hervorragend für Betrugserkennung, Empfehlungen und Netzwerkanalyse und sind meist ein schlechter Primärspeicher für transaktionale Geschäftsdaten.

Zeitreihen-, Such- und Vektorspeicher

Eine Zeitreihendatenbank ist für Messwerte mit Zeitstempel optimiert, die der Reihe nach eintreffen und selten aktualisiert werden: Metriken, Sensorwerte, Preise, Standortverläufe. Sie komprimiert stark, partitioniert nach Zeit, verdichtet alte Daten (jeden Punkt eine Woche lang behalten, Minutenmittelwerte ein Jahr lang) und löscht abgelaufene Daten günstig, indem sie ganze Zeitpartitionen verwirft. Beispiele sind InfluxDB, Prometheus (für Monitoring) und TimescaleDB, eine PostgreSQL-Erweiterung, die zeigt, wie oft sich eine relationale Datenbank passend dehnen lässt.

Eine Suchmaschine wie Elasticsearch, OpenSearch oder Apache Solr (alle auf der Bibliothek Apache Lucene aufgebaut) pflegt einen *invertierten Index* von Wörtern zu den Dokumenten, die sie enthalten, mit Textanalyse (Stammformreduktion, Synonyme, Tippfehlertoleranz) und Relevanzranking. Sie beantwortet „veganes Ramen in meiner Nähe“ in Millisekunden. Sie sollte fast nie ein System of Record sein: Indizes werden nahezu in Echtzeit statt transaktional aufgefrischt, und das übliche Design speist die Suchmaschine aus der besitzenden Datenbank und baut den Index bei Bedarf daraus neu auf.

Eine Vektordatenbank speichert *Embeddings*, Zahlenlisten, die die Bedeutung von Texten, Bildern oder Produkten darstellen, und findet die Elemente, die einem Abfragevektor am nächsten liegen. Eine exakte Suche vergleicht die Abfrage mit jedem Vektor, deshalb nutzen diese Speicher Indizes für approximative Nächste-Nachbarn-Suche wie HNSW-Graphen und tauschen ein wenig Trefferquote gegen Geschwindigkeit. Sie treiben semantische Suche, Empfehlungen und das Retrieval für große Sprachmodelle an, behandelt in Retrieval-

Augmented Generation. Es gibt eigene Produkte (Milvus, Qdrant, Pinecone, Weaviate), aber für Millionen statt Milliarden von Vektoren genügt oft eine Erweiterung einer bestehenden Datenbank wie pgvector.

Distributed SQL

Distributed-SQL-Datenbanken, manchmal *NewSQL* genannt (ein Begriff, den der Analyst Matthew Aslett 2011 prägte), behalten das relationale Modell, SQL und ACID-Transaktionen bei, partitionieren und replizieren Daten aber automatisch über viele Knoten, oft über Regionen hinweg. Google Spanner, beschrieben in einem OSDI-Paper von 2012, nutzt eng synchronisierte Uhren (seine TrueTime-API, gestützt auf GPS und Atomuhren), um global konsistente Transaktionen zu bieten. CockroachDB, YugabyteDB und TiDB bieten ähnliche Modelle auf gewöhnlicher Hardware; CockroachDB nutzt standardmäßig serialisierbare Isolation.

Der Preis sind Physik und Betrieb. Eine Transaktion, der Replikas in mehreren Regionen zustimmen müssen, wartet auf einen Konsens-Roundtrip, sodass einzelne Zeilenschreibvorgänge Millisekunden bis zu Dutzenden oder Hunderten Millisekunden dauern statt deutlich unter einer Millisekunde. Abfrageplanung über Partitionen hinweg ist subtiler, die Kosten sind höher, und weniger Fachleute kennen die Fehlerbilder. Distributed SQL ist das richtige Werkzeug, wenn Sie wirklich relationale Transaktionen jenseits der Kapazität eines Servers oder über Regionen hinweg brauchen (ein globales Zahlungshauptbuch, eine Bank in mehreren Regionen), und ein teures, wenn ein einzelner PostgreSQL-Primary mit Replikas genügen würde.

Vergleich der Datenbankfamilien

Klasse	Datenmodell und Abfrage	Stärken	Schwächen	Beispiele
Relational	Tabellen, Schlüssel, SQL mit Joins und Aggregationen	Transaktionen und Constraints, flexible Ad-hoc-Abfragen, ausgereifte Werkzeuge, weithin bekannt	Vertikale Skalierungsgrenze für Schreibvorgänge auf einem Knoten; starres Schema erfordert Migrationen	PostgreSQL, MySQL, SQL Server, Oracle
Dokument	JSON-ähnliche Dokumente, Abfragen auf Felder und verschachtelte Werte	Natürliche Passform für Aggregate, die als Ganzes gelesen werden; flexible Struktur	Joins und dokumentübergreifende Transaktionen schwächer oder teurer; Schema im Code durchgesetzt	MongoDB, Couchbase, Firestore
Key-Value	Opaker Wert pro Schlüssel; get, put, delete	Sehr schnell, einfach, leicht zu partitionieren	Keine Abfragen über Werte; Modellierungslast bei der Anwendung	Redis, Valkey, DynamoDB, etcd
Wide-Column	Zeilen nach Partition Key gruppiert, nach Clustering Key sortiert	Enormer Schreibdurchsatz, lineare horizontale Skalierung, mehrere Rechenzentren	Tabellen pro Abfrage gebaut; begrenzte Joins und Ad-hoc-Abfragen	Apache Cassandra, ScyllaDB, HBase, Bigtable
Graph	Knoten und Kanten mit Eigenschaften; Traversierungsabfragen	Beziehungsabfragen über mehrere Schritte	Schwerer zu partitionieren; kleineres Ökosystem	Neo4j, Amazon Neptune
Zeitreihen	Punkte mit Zeitstempel, Tags; Bereichs- und Verdichtungsabfragen	Kompression, Aufbewahrungsfristen, Downsampling	Schwach bei allgemeinen Aktualisierungen und Beziehungen	InfluxDB, TimescaleDB, Prometheus
Suchmaschine	Invertierter Index über Dokumente; relevanzgewichtete Textabfragen	Volltextsuche, unscharfe Suche, Facetten	Kein System of Record; Auffrischung nahezu in Echtzeit	Elasticsearch, OpenSearch, Apache Solr

Klasse	Datenmodell und Abfrage	Stärken	Schwächen	Beispiele
Vektor	Embeddings; approximative Nächste-Nachbarn-Suche	Semantische Ähnlichkeit für Retrieval und Empfehlungen	Näherungsweise Ergebnisse; Kosten für den Indexaufbau; junge Werkzeuge	pgvector, Milvus, Qdrant, Pinecone
Distributed SQL	Relationales Modell, SQL, partitioniert und repliziert	Transaktionen und SQL mit horizontaler Skalierung und Überleben über Regionen hinweg	Höhere Schreiblatenz, Kosten und betriebliche Komplexität	Google Spanner, CockroachDB, YugabyteDB, TiDB

Die Wahl: Kriterien, die zählen

Wählen Sie pro Datenbestand, nicht pro System. Sammeln Sie für jeden wichtigen Datenbestand Belege zu sechs Kriterien und halten Sie die Entscheidung in einem ADR fest.

Datenform. Sind die Daten von Natur aus tabellarisch mit vielen Beziehungen, ein in sich geschlossenes Aggregat, ein Netzwerk, eine Zeitreihe, freier Text oder Vektoren? **Zugriffsmuster.** Welche Abfragen laufen, wie oft, mit welchen Latenzzielen, und wie viele davon sind ad hoc? Überwiegend lesend oder schreibend? Immer per Schlüssel oder nach vielen Attributen? **Konsistenzbedarf.** Brauchen die Daten Transaktionen über mehrere Elemente und Constraints (Kontostände, Lagerbestand, Buchungen), oder sind ein paar Sekunden Veralterung harmlos (Likes, Empfehlungen)? Siehe Konsistenzmodelle. **Größenordnung.** Aktuelle und prognostizierte Datenmenge, Schreibvorgänge pro Sekunde und Wachstum, gemessen, nicht geraten; ein Speicher, der auf Petabytes skaliert, ist bei 50 Gigabyte kein Vorteil. **Betrieb.** Wer wird ihn betreiben: Backups, Upgrades, Failover, Monitoring, Sicherheitspatches? Gibt es einen Managed Service in Ihrer Cloud, und was kostet er bei Ihren Volumina? **Teamkenntnisse.** Was kennt das Team schon gut genug, um es um 3 Uhr nachts zu debuggen? Unvertraute Technologie bringt unbekannte Fehlerbilder mit sich.

Zwei Gewohnheiten verbessern die Wahl. Schreiben Sie die fünf oder zehn wichtigsten Abfragen mit ihren Ziellatenzen *vor* der Wahl auf, und testen Sie den Kandidaten mit realistischen Datenmengen; viele Enttäuschungen stammen von einem Speicher, der auf einem Laptop mit tausend Zeilen getestet wurde. Und trennen Sie das *System of Record* von *abgeleiteten Speichern*: Das System of Record braucht die stärksten Garantien und die konservativste Technologie, während abgeleitete Speicher (Suchindizes, Caches, Read Models) gerade deshalb spezialisiert sein können, weil sie sich neu aufbauen lassen.

ENTSCHEIDUNGSHILFE

Entscheidungshilfe: Welche Art von Speicher für diese Daten?

Stellen Sie diese Fragen für jeden Datenbestand (nicht einmal für das ganze System), in dieser Reihenfolge. Halten Sie beim ersten passenden *Ja* an; erreichen Sie das Ende, bleibt es beim relationalen Standard.

Q1 **Sind diese Daten ein System of Record, das Transaktionen über mehrere Zeilen, Constraints oder Ad-hoc-Abfragen braucht (Bestellungen, Zahlungen, Konten, Buchungen)?**

Ja Nutzen Sie eine relationale Datenbank. Suchen Sie nur weiter, wenn Sie zeigen können, dass sie eine gemessene Anforderung an Größenordnung oder Latenz nicht erfüllen kann.

Nein Weiter: Die Daten sind womöglich abgeleitet, spezialisiert oder haben extrem hohes Volumen.

Q2 **Ist es eine aus einem System of Record abgeleitete Kopie, die eine spezielle Abfrage bedient (Volltextsuche, Empfehlungen, Ähnlichkeitssuche)?**

Ja Nutzen Sie für diese Abfrage den spezialisierten Speicher (Suchmaschine, Graph- oder Vektorindex), gespeist aus dem System of Record und daraus neu aufbaubar.

Nein Weiter.

Q3 **Wird es als einzelnes Element per Schlüssel gelesen und geschrieben, ohne Abfragen über Elemente hinweg (Sessions, Warenkörbe, letzte Positionen, Feature-Flags)?**

Ja Ein Key-Value-Store passt, oft im Arbeitsspeicher mit Ablaufzeit. Legen Sie fest, was geschieht, wenn er verloren geht.

Nein Weiter.

Q4 **Ist es ein überwiegend anhängender Strom von Messwerten mit Zeitstempel, der nach Zeitbereich abgefragt wird (Metriken, Sensorwerte, Standortverlauf)?**

Ja Eine Zeitreihendatenbank oder, bei moderaten Volumina, zeitpartitionierte Tabellen in einer relationalen Datenbank.

Nein Weiter.

Q5 **Werden Schreibvolumen oder Datenmenge übersteigen, was ein gut abgestimmter Server bewältigen kann, bei Zugriff immer über einen bekannten Schlüssel?**

Ja Ein Wide-Column- oder Key-Value-Store, der für Partitionierung ausgelegt ist (Cassandra, DynamoDB), mit Tabellen, die um die Abfragen herum entworfen sind.

Nein Weiter.

Q6 **Brauchen Sie relationale Transaktionen und SQL über Regionen hinweg oder jenseits der Kapazität eines Servers?**

Ja Erwägen Sie Distributed SQL (Spanner, CockroachDB, YugabyteDB) und nehmen Sie höhere Schreiblatenz und Betriebskosten in Kauf.

Nein Bleiben Sie beim relationalen Standard und prüfen Sie erneut, wenn sich eine gemessene Anforderung ändert.

WENN DAS NEU FÜR SIE IST

Schritt für Schritt: die einzige Kopie oder eine Kopie

Die Idee, die hier die meiste Sorgfalt verlangt, ist der Unterschied zwischen einem System of Record und einem abgeleiteten Speicher und warum er bestimmt, wie experimentierfreudig eine Datenbankwahl sein darf. Verfolgen Sie eine Speisekartenänderung durch die Speicher der Essenslieferplattform.

- Ein Restaurant ändert die Beschreibung eines Gerichts, und die Änderung wird in der PostgreSQL-Datenbank für Speisekarten gespeichert. Diese Datenbank ist das System of Record: Wann immer eine andere Kopie einer Speisekarte von ihr abweicht, gilt ihre Version.
- Change Data Capture greift die committete Änderung auf und gibt sie an die Suchmaschine weiter, die ihren Index einige Sekunden später aktualisiert. In diesen Sekunden kann eine Suche noch die alte Beschreibung zeigen, eine Verzögerung, die das Team akzeptiert hat.
- Nehmen Sie nun an, der Suchcluster fällt aus und sein Index geht verloren. Kunden können eine Weile nicht suchen, aber keine Information ist verloren: Das Team baut den Index neu auf, indem es jede Speisekarte erneut aus PostgreSQL liest.
- Stellen Sie sich stattdessen vor, die Änderungen der Restaurants würden nur in der Suchmaschine gespeichert. Ihr Index wird nahezu in Echtzeit statt innerhalb von Transaktionen aufgefrischt, und sie schützt jeden Schreibvorgang womöglich weniger sorgfältig als die Datenbank, sodass derselbe Ausfall Änderungen verlieren könnte, ohne dass es etwas gäbe, woraus man sie wiederherstellen könnte.
- Deshalb kann die Plattform für die Suche einen spezialisierten Speicher betreiben, gewählt für genau diese eine Aufgabe, während Bestellungen und Speisekarten auf PostgreSQL bleiben, der Datenbank, die das Team bereits gut betreibt.
- **Die Lehre:** Fragen Sie vor der Wahl eines Speichers, ob er die einzige Kopie von irgendetwas halten wird. Wenn ja, wählen Sie die konservativste Technologie, die Sie können; hält er eine Kopie, die sich neu aufbauen lässt, dürfen Sie sich spezialisieren.

Polyglotte Persistenz und ihre Kosten

Polyglotte Persistenz bedeutet, innerhalb eines Systems unterschiedliche Speichertechnologien für unterschiedliche Daten zu nutzen, gewählt danach, wie die Daten verwendet werden. Martin Fowler machte den Begriff 2011 bekannt und schrieb seine erste Verwendung Scott Leberknight zu. Er folgt natürlich aus Datenbank pro Service: Jeder Eigentümer kann das richtige Werkzeug wählen.

Die Kosten werden leicht unterschätzt. Jeder zusätzliche Datenbanktyp braucht eigene Expertise, eigene Verfahren für Backup und Wiederherstellung, Monitoring, Kapazitätsplanung, Sicherheitshärtung, einen eigenen Upgrade-Zyklus und Wissen in der Rufbereitschaft. Daten bewegen sich nun zwischen Speichern, also brauchen Sie Pipelines (CDC, Events) und müssen Verzögerung und Ausfall jeder einzelnen bedenken. Konsistenz zwischen Speichern ist bestenfalls eventual. Für ein kleines Team können fünf Datenbanken mehr Aufwand verschlingen als das Produkt. Eine vernünftige Regel ist, jeden neuen Speicher durch eine Anforderung zu begründen, die die bestehenden nachweislich nicht erfüllen können, und für die spezialisierten Managed Services zu bevorzugen.

WOHER DAS KOMMT

Choose Boring Technology

In einem Essay von 2015 argumentierte Dan McKinley, gestützt auf seine Jahre als Engineer bei Etsy, dass jedes Unternehmen nur wenige *Innovation Tokens* bekommt, etwa drei, die es für unvertraute Technologie ausgeben kann. Ausgereifte, „langweilige“ Werkzeuge haben bekannte Fehlerbilder; neue bringen unbekannte Unbekannte mit sich. Jede neue Technologie fügt der gesamten Organisation außerdem dauerhafte betriebliche und kognitive Kosten hinzu. Sein Rat: Fragen Sie zuerst, wie Sie das Problem mit dem lösen würden, was Sie bereits betreiben, und behandeln Sie das Hinzufügen einer Technologie als Entscheidung der Organisation, nicht eines einzelnen Teams. Der Essay wird bei Datenbankentscheidungen viel zitiert, wo die Kosten für den Betrieb von etwas Unvertrautem jahrelang anhalten.

AUSGEARBEITETES BEISPIEL

Ausgearbeitetes Beispiel: Speicher für die Essenslieferplattform

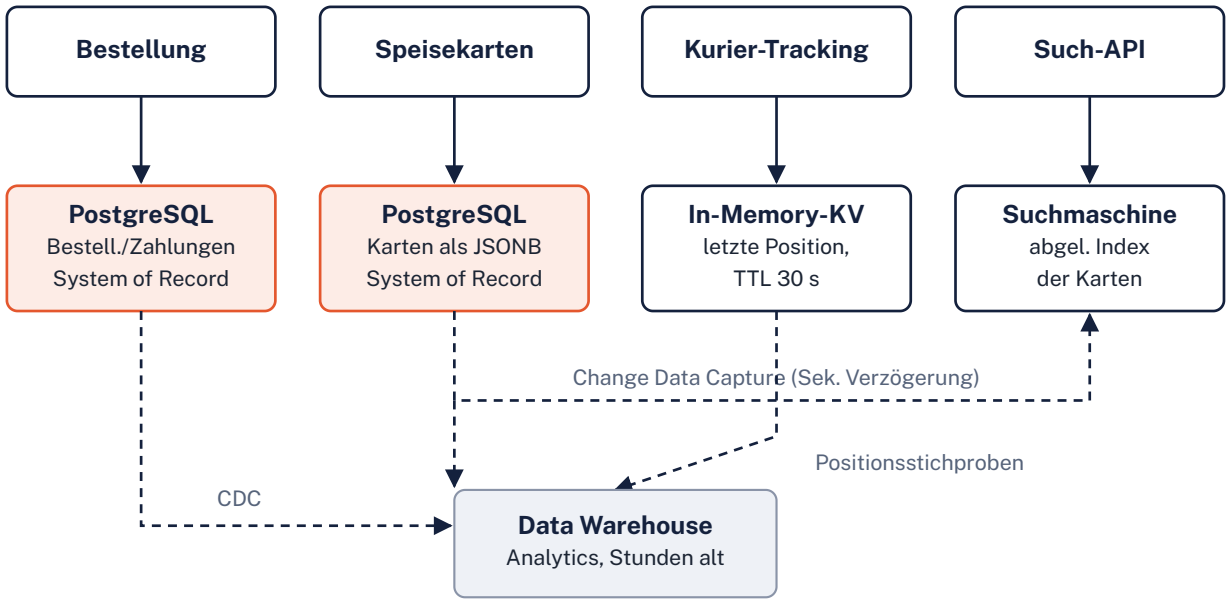
Die Plattform ist in 40 Städten tätig. In der Abendspitze nimmt sie etwa 300 Bestellungen pro Minute an, hat 20.000 Kuriere online, liefert 50.000 Restaurant-Speisekarten aus und wird etwa 2000-mal pro Sekunde durchsucht. Das Team aus 35 Entwicklungsfachleuten betreibt PostgreSQL bereits gut.

Bestellungen und Zahlungen	System of Record; Transaktionen über mehrere Zeilen (Bestellung, Positionen, Zahlungsautorisierung); strenge Regeln; Ad-hoc-Abfragen durch den Support. Etwa 5 neue Bestellungen pro Sekunde in der Spitze (ein paar Dutzend Zeilenschreibvorgänge). **PostgreSQL** mit einem Replikat für Failover. Kein Grund für etwas anderes.
Speisekarten	Überwiegend lesend, hierarchisch (Rubriken, Gerichte, Optionsgruppen), von Restaurants einige tausend Mal am Tag bearbeitet. Preise müssen genau dem entsprechen, was die Bestellung festhält. **PostgreSQL** mit einem relationalen Kern (Restaurant, Gericht, Preis) und `jsonb` für variable Optionen. Ein Dokumentenspeicher würde zur Form passen, aber eine zweite Technologie hinzufügen, ohne eine Anforderung zu erfüllen, die PostgreSQL verfehlt.
Kurierstandorte	20.000 Kuriere, die jeweils alle 4 Sekunden eine Position senden, ergeben 5000 Schreibvorgänge pro Sekunde, von denen für die Disposition nur die jeweils letzte pro Kurier zählt, und das nur etwa 30 Sekunden lang. **In-Memory-Key-Value-Store** (Redis oder Valkey) mit Geoabfragen und 30 Sekunden Ablaufzeit. Ihn bei einem Neustart zu verlieren kostet einen Aktualisierungszyklus. Eine Stichprobe von Punkten (etwa einer alle 30 Sekunden) geht für Analytics ins Warehouse.
Suche nach Restaurants und Gerichten	Volltext, tippfehlertolerant, gefiltert nach Standort, Öffnungszeiten und Ernährungs-Tags, nach Relevanz gerankt. **Suchmaschine** (OpenSearch), gespeist aus der Speisekarten-Datenbank per Change Data Capture und daraus neu aufbaubar. Akzeptiert nach einer Speisekartenänderung einige Sekunden Verzögerung.
Analytics	Wochenumsatz, Lieferzeiten nach Stadt, Auslastung der Kuriere: große Scans und Aggregationen über Monate an Historie. **Cloud-Data-Warehouse**, gespeist durch CDC und Events, konstruktionsbedingt Stunden alt. Siehe Analytics und Streaming.
Verworfen	Eine Graphdatenbank für Empfehlungen (noch nicht nötig; mit Belegen neu prüfen), eine separate Zeitreihendatenbank für Standorte (die Stichprobe im Warehouse beantwortet die gestellten Fragen) und Distributed SQL (die Bestellungen der gesamten Plattform, etwa 5 pro Sekunde in der Spitze, passen bequem auf einen Primary; es gibt keine Anforderung an regionsübergreifende Transaktionen).

Das Ergebnis sind zwei Systems of Record auf einer bewährten Technologie plus drei spezialisierte, abgeleitete Speicher. Jede zusätzliche Technologie ist durch eine Anforderung begründet, die der Standard nicht erfüllen kann: 5000 kurzlebige Schreibvorgänge pro Sekunde und relevanzgerankte Textsuche.

Die Abfragen und Zahlen zuerst aufzuschreiben machte die Entscheidungen fast offensichtlich und gab dem Team eine Aufzeichnung der Annahmen, die es neu prüfen kann, wenn sich die Volumina ändern.

Die resultierende Speicherlandschaft



Zwei Systems of Record; alles andere ist abgeleitet und neu aufbaubar

Bestellungen und Speisekarten haben je ein einziges relationales System of Record; Suchindex, Warehouse und Standortspeicher sind abgeleitet oder verzichtbar.

Häufige Fehler

Lebenslaufgetriebene Entscheidungen: einen Speicher wählen, weil er neu ist oder weil ein großes Unternehmen ihn nutzt, ohne die Anforderung, die ihn dort rechtfertigte. **Für eingebildete Größenordnungen wählen**: für eine Milliarde Nutzer entwerfen bei einem Produkt mit tausend und vom ersten Tag an die Komplexitätssteuer zahlen. **Einen abgeleiteten Speicher als System of Record nutzen**: ein Suchindex oder Cache, der zur einzigen Kopie von Daten wird, mit schwächerer Dauerhaftigkeit, als irgendjemand ahnte. **Die Abfrageseite ignorieren**: einen Key-Value- oder Wide-Column-Store wählen und dann feststellen, dass das Geschäft Ad-hoc-Berichte und Filter braucht, die er nicht bedienen kann. **Den Betrieb**

unterschätzen: vergessen, dass jemand ihn aus dem Backup wiederherstellen, ohne Ausfallzeit upgraden und unter Last diagnostizieren muss. Und schließlich **die Wahl als endgültige Wahrheit behandeln**: Halten Sie die Annahmen fest (Volumina, Abfragen, Teamkenntnisse), damit die Entscheidung neu geprüft werden kann, wenn sie sich ändern.

PRÜFEN SIE IHR VERSTÄNDNIS

▼ **Warum sollte eine Suchmaschine in der Regel nicht das System of Record sein?**

Ihr Index wird nahezu in Echtzeit statt transaktional aufgefrischt, und ihre Dauerhaftigkeitsgarantien sind schwächer; sie sollte aus der besitzenden Datenbank gespeist und bei Bedarf daraus neu aufgebaut werden.

▼ **Nennen Sie zwei Anzeichen dafür, dass ein Wide-Column-Store gut passen könnte.**

Sehr hohes Schreibvolumen jenseits eines Servers und ein Zugriff, der immer bei einem bekannten Partition Key ansetzt, sodass sich Tabellen pro Abfrage entwerfen lassen.

▼ **Wofür steht ein „Innovation Token“ in McKinleys Essay?**

Für eine von wenigen Gelegenheiten, die ein Unternehmen hat, unvertraute Technologie einzuführen, die jeweils unbekannte Fehlerbilder und dauerhafte Betriebskosten mit sich bringt.

▼ **Wann ist Distributed SQL seine Kosten wert?**

Wenn Sie relationale Transaktionen und SQL jenseits der Kapazität eines Servers oder über Regionen hinweg brauchen, etwa für ein globales Hauptbuch; nicht, wenn ein einzelner Primary mit Replikas die Anforderungen erfüllt.

▼ **Probieren Sie es aus: Ein Team von 20 Personen bei einem Essenslieferdienst kennt PostgreSQL gut. Es braucht Speicher für Bestellungen und Zahlungen, für Live-Positionen der Kuriere, die alle paar Sekunden aktualisiert werden, und für die Textsuche nach Restaurants. Wählen Sie jeweils einen Speicher, und sagen Sie, was begrenzt, wie viele Sie hinzufügen.**

Bestellungen und Zahlungen bleiben in PostgreSQL: ein System of Record, das Transaktionen, Constraints und Ad-hoc-Abfragen braucht. Kurierpositionen passen zu einem Key-Value-Store mit kurzer Ablaufzeit, da nur der letzte Wert zählt und er sich schnell ändert. Die Restaurantsuche passt zu einer Suchmaschine oder anfangs zur Textsuche von PostgreSQL, als abgeleiteter Speicher, der neu aufgebaut werden kann. Jede weitere Technologie verursacht dauerhafte Betriebskosten, schreiben Sie also Abfragen und Volumen auf, und ergänzen Sie eine nur auf der Grundlage von Belegen.

QUELLEN

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

Transaktionen, ACID und Isolationsebenen

BEVOR SIE DAS LESEN

Dieser Abschnitt baut auf:

- 0.5 Daten speichern

Begriffe von früher, die er verwendet: SQL (Structured Query Language), Fremdschlüssel, Index, Linearisierbarkeit

DIE IDEE IN KÜRZE

Stellen Sie sich zwei Kassierer vor, die sich ein Papierkontobuch für das Konto eines Kunden teilen. Jeder schlägt den Kontostand nach, rechnet den neuen auf einem Notizblock aus und trägt ihn ein. Tun sie das im selben Moment, geht die Änderung eines Kassierers verloren. Eine Datenbanktransaktion ist eine Regel dafür, wie die Kassierer sich abwechseln: ob sie in die halb geschriebenen Einträge des anderen schauen dürfen, ob sich die Seite ändern kann, während sie sie lesen, und ob einer auf den anderen warten muss. Strengere Regeln bedeuten weniger Fehler, aber mehr Warten, und die Namen dieser Regeln bedeuten in verschiedenen Banken leicht Unterschiedliches.

WAS SIE WISSEN MÜSSEN

- ACID: Atomarität bedeutet Alles-oder-nichts im Fehlerfall; Konsistenz bedeutet, dass die Invarianten der Anwendung gelten; Isolation begrenzt die Beeinflussung zwischen nebenläufigen Transaktionen; Dauerhaftigkeit bedeutet, dass festgeschriebene Daten Abstürze überstehen, meist durch ein Write-Ahead-Log und Replikate.
- Der SQL-Standard definiert Read Uncommitted, Read Committed, Repeatable Read und Serializable über die Dirty Reads, Non-Repeatable Reads und Phantome, die sie verbieten; Berenson et al. (1995) zeigten, dass diese Definitionen mehrdeutig und unvollständig sind.
- Lost Updates und Write Skew entstehen, wenn Transaktionen Daten lesen, in der Anwendung entscheiden und zurückschreiben; Snapshot Isolation verhindert Lost Updates, lässt aber Write Skew zu.
- MVCC hält mehrere Versionen einer Zeile vor, sodass Leser und Schreiber einander nicht blockieren; lang laufende Transaktionen blähen den Speicher auf.
- Standardwerte und Namen unterscheiden sich: PostgreSQL und Oracle verwenden standardmäßig Read Committed, MySQL InnoDB Repeatable Read, CockroachDB Serializable; Oracles Serializable ist Snapshot Isolation.

- Schützen Sie Read-Modify-Write-Pfade mit Constraints, atomaren bedingten Updates, SELECT … FOR UPDATE (pessimistisch) oder Versionsspalten (optimistisch), oder verwenden Sie Serializable mit Wiederholung der ganzen Transaktion.

Was eine Transaktion verspricht

Eine Transaktion fasst mehrere Lese- und Schreibvorgänge zu einer Einheit zusammen, die entweder vollständig wirksam wird (Commit) oder gar nicht (Abbruch und Rollback). Kapitel 0 hat die Idee mit einer Banküberweisung eingeführt. In diesem Abschnitt geht es um den Teil, den die meisten Entwicklungsteams missverstehen: was passiert, wenn viele Transaktionen gleichzeitig laufen, welche Garantien Datenbanken wirklich geben und wie man wählt. Die Beispiele verwenden SQL, aber die Überlegungen gelten für jeden Speicher, der Transaktionen anbietet.

Transaktionen sind architektonisch wichtig, weil sie entscheiden, wo sich Geschäftsregeln günstig durchsetzen lassen. Innerhalb einer Datenbank kann eine Transaktion mit wenigen Zeilen Code ein Kontobuch ausgeglichen halten oder verhindern, dass ein Tisch doppelt gebucht wird. Über Services hinweg gibt es kein solches Werkzeug, weshalb Datenhoheit und Sagas so wichtig sind. Wer genau weiß, was eine einzelne Datenbank garantiert, weiß auch, welche Regeln innerhalb der Grenze eines einzigen Eigentümers liegen sollten.

ACID im Detail

ACID wurde 1983 von Theo Härder und Andreas Reuter geprägt, aufbauend auf früheren Arbeiten von Jim Gray. Jeder Buchstabe ist enger gefasst, als er klingt. **Atomarität** betrifft Fehler, nicht Nebenläufigkeit: Geht unterwegs etwas schief (ein Absturz, eine Constraint-Verletzung, ein Deadlock, die Anwendung entscheidet sich zum Abbruch), wird jede bis dahin vorgenommene Änderung rückgängig gemacht, sodass die Anwendung gefahrlos wiederholen kann. *Abbrechbarkeit* wäre ein treffenderer Name.

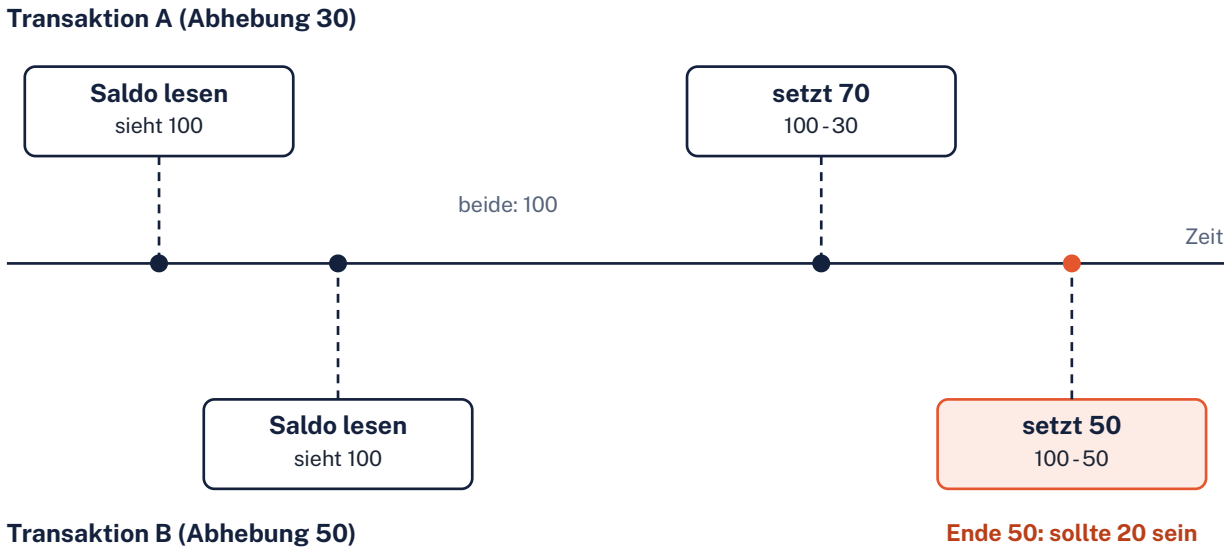
Konsistenz bedeutet, dass eine Transaktion die Datenbank von einem gültigen Zustand in einen anderen überführt, wobei „gültig“ heißt, dass Ihre Invarianten gelten: Soll gleich Haben, der Bestand ist nie negativ, jede Bestellung hat einen Kunden. Manche Invarianten kann die Datenbank selbst durchsetzen (Schlüssel, Fremdschlüssel, `CHECK`-Constraints, eindeutige Indizes), die meisten aber sind Geschäftsregeln, die nur die Transaktionen der Anwendung einhalten. Konsistenz ist daher eine Eigenschaft davon, wie die Anwendung Atomarität und Isolation nutzt, nicht etwas, das die Datenbank allein liefert. Sie hat auch nichts mit dem „C“ in CAP zu tun, das Linearisierbarkeit bedeutet (Kapitel 6).

Isolation bedeutet, dass gleichzeitig laufende Transaktionen einander nicht stören. In ihrer strengsten Form, der Serialisierbarkeit, ist das Ergebnis nebenläufig ausgeführter Transaktionen dasselbe, als wären sie in irgendeiner Reihenfolge nacheinander gelaufen. Schwächere Ebenen erlauben im Tausch gegen Performance eine gewisse Beeinflussung, und der Rest dieses Abschnitts handelt genau davon, welche. **Dauerhaftigkeit** bedeutet, dass die Daten Abstürze überstehen, sobald ein Commit bestätigt ist. Datenbanken auf einem Knoten erreichen das mit einem Write-Ahead-Log, das auf die Platte geschrieben wird, bevor der Commit zurückkehrt; replizierte Datenbanken fügen Kopien auf anderen Maschinen hinzu. Dauerhaftigkeit ist nie absolut: Eine Platte kann ein Flush nur vortäuschen, und asynchrone Replikation kann die letzten bestätigten Schreibvorgänge verlieren, wenn der Leader ausfällt (siehe Replikation).

Was ohne Isolation schiefgeht

Nebenläufigkeitsfehler sind schwer zu finden, weil sie vom Timing abhängen: Sie bestehen jeden Test auf einem ruhigen Entwicklerrechner und tauchen zur Spitzenzeit am Abend auf. Am klarsten versteht man Isolationsebenen über die konkreten Anomalien (auch Phänomene genannt), die sie zulassen oder verhindern. Das Lost Update unten ist im Geschäftscode die häufigste.

Ein Lost Update, Schritt für Schritt



Jede Transaktion ist für sich korrekt; verschränkt löscht der zweite Schreibvorgang stillschweigend die erste Abhebung.

Die Anomalien, von der mildesten bis zur subtilsten

Anomalie	Was passiert	Beispiel aus Essenslieferung oder Bank
Dirty Write	Eine Transaktion überschreibt Daten, die eine andere Transaktion geschrieben, aber noch nicht festgeschrieben hat.	Zwei Transaktionen, die einen Kurier zuweisen, verschränken sich so, dass die Bestellzeile einen Kurier nennt und die Kurierzeile eine andere Bestellung. Jede echte Isolationsebene verhindert das.
Dirty Read	Eine Transaktion liest Daten, die eine andere Transaktion geschrieben, aber nicht festgeschrieben hat (und die womöglich zurückgerollt werden).	Ein Bericht zählt eine Zahlung über 500, die anschließend zurückgerollt wird.
Non-Repeatable (Fuzzy) Read	Eine Transaktion liest dieselbe Zeile zweimal und erhält unterschiedliche Werte, weil eine andere Transaktion zwischendurch eine Änderung festgeschrieben hat.	Der Checkout liest einen Gerichtpreis von 9,50, eine Änderung des Restaurants schreibt 11,00 fest, und der zweite Lesevorgang des Checkouts sieht 11,00.
Read Skew	Eine Transaktion sieht Teile der Datenbank zu unterschiedlichen Zeitpunkten, sodass zusammengehörige Werte nicht aufgehen.	Ein Kunde mit 500 auf zwei Konten sieht insgesamt 450, weil zwischen dem Lesen der beiden Salden eine Überweisung von 50 festgeschrieben wurde.
Phantom Read	Eine Transaktion führt eine Abfrage mit einer Bedingung erneut aus und findet Zeilen, die eine andere, festgeschriebene Transaktion hinzugefügt oder entfernt hat.	Zweimaliges Zählen der offenen Lieferungen eines Kuriers liefert erst 2, dann 3.
Lost Update	Zwei Transaktionen lesen einen Wert, jede berechnet einen neuen und schreibt ihn, und ein Schreibvorgang überschreibt stillschweigend den anderen.	Zwei Abhebungen vom selben Konto (Diagramm oben).
Write Skew	Zwei Transaktionen lesen überlappende Daten, treffen darauf gestützt Entscheidungen und ändern verschiedene Zeilen, sodass sie gemeinsam eine Regel verletzen, die jede für sich geprüft hat.	Zwei Ärzte in Bereitschaft sehen jeweils, dass noch jemand anderes Bereitschaft hat, und beenden beide ihre Bereitschaft (ausgearbeitetes Beispiel unten).

Die Isolationsebenen des SQL-Standards

Der SQL-Standard (ab SQL-92) definiert vier Isolationsebenen danach, welche von drei Phänomenen sie verbieten: Dirty Reads, Non-Repeatable Reads und Phantome. Der Standard macht Serializable zur Voreinstellung, doch kaum ein gängiges Produkt hält sich daran; die meisten verwenden aus Performancegründen standardmäßig Read Committed oder Repeatable Read.

Die Ebenen des SQL-Standards und die Phänomene, die sie ausschließen

Isolationsebene	Dirty Read	Non-Repeatable Read	Phantom Read
Read Uncommitted	möglich	möglich	möglich
Read Committed	verhindert	möglich	möglich
Repeatable Read	verhindert	verhindert	möglich
Serializable	verhindert	verhindert	verhindert, und das Ergebnis muss einer seriellen Reihenfolge entsprechen

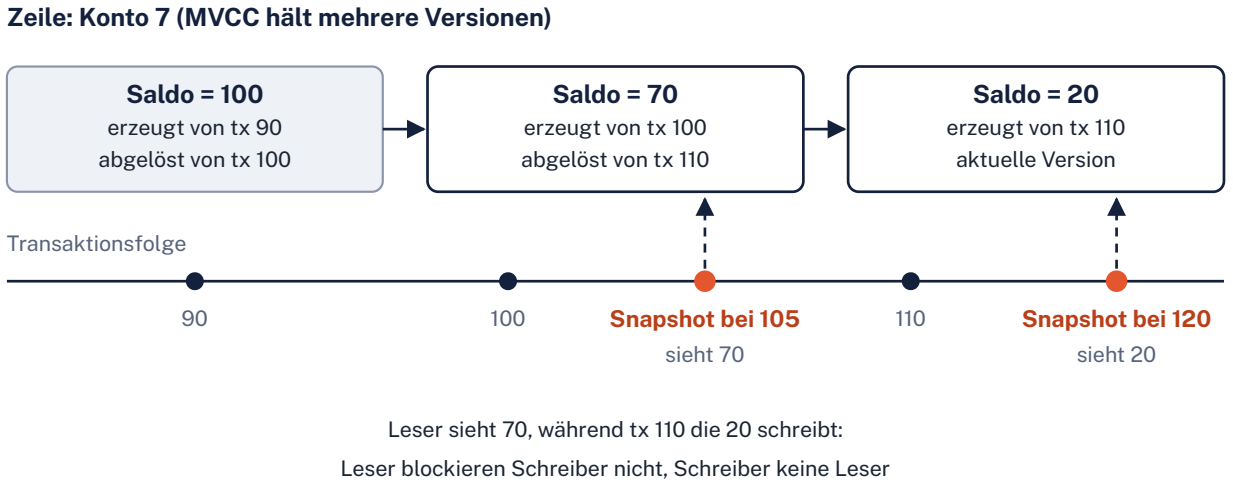
1995 veröffentlichten Hal Berenson, Phil Bernstein, Jim Gray, Jim Melton, Elizabeth O'Neil und Patrick O'Neil *A Critique of ANSI SQL Isolation Levels*. Sie zeigten, dass die englischsprachigen Definitionen des Standards mehrdeutig waren, dass das Verbot der drei Phänomene allein noch keine Serialisierbarkeit ergibt und dass die Ebenen des Standards die Isolationsebenen realer Produkte nicht beschrieben. Sie ergänzten Phänomene, die der Standard übersehen hatte, namentlich Dirty Write, Lost Update, Read Skew und Write Skew, und definierten Snapshot Isolation: Jede Transaktion liest aus einem konsistenten Snapshot der festgeschriebenen Daten zum Zeitpunkt ihres Starts, und eine Transaktion kann keinen Schreibvorgang auf eine Zeile festschreiben, die eine nebenläufige Transaktion bereits geändert und festgeschrieben hat („First Committer Wins“). Snapshot Isolation verhindert Dirty Reads, Non-Repeatable Reads, Read Skew und Lost Updates sowie die Phantome, die Lesevorgänge betreffen, lässt aber Write Skew zu und ist daher nicht serialisierbar. Sie passt nur schlecht zwischen die Ebenen des Standards, weshalb Produkte sie unterschiedlich benennen.

MVCC: wie Snapshots funktionieren

Die meisten modernen Datenbanken setzen Snapshot-Lesevorgänge mit Multi-Version Concurrency Control (MVCC) um. Statt eine Zeile an Ort und Stelle zu überschreiben, erzeugt ein Schreibvorgang eine neue Version, die mit der Kennung der schreibenden Transaktion

versehen ist, und die alte Version bleibt erhalten, bis keine laufende Transaktion sie mehr brauchen kann. Jeder Leser erhält einen Snapshot, im Wesentlichen eine Liste der Transaktionen, die bei seinem Start festgeschrieben waren, und sieht nur die Versionen, die diese Transaktionen erzeugt haben. Der große Vorteil: Leser warten nie auf Schreiber und Schreiber nie auf Leser; nur zwei Schreiber auf dieselbe Zeile geraten in Konflikt.

Zeilenversionen unter MVCC



Jede Änderung erzeugt eine neue Zeilenversion; eine Transaktion sieht die neueste Version, die vor ihrem Snapshot festgeschrieben wurde.

Der Preis ist Aufräumarbeit. Alte Versionen müssen entfernt werden: PostgreSQL hält sie in der Tabelle und entfernt sie mit *Vacuum*, Oracle und MySQL InnoDB halten Before-Images in Undo-Logs. Eine stundenlang offene Transaktion hält alte Versionen fest, sodass Tabellen und Undo-Logs anwachsen und die Performance sinkt. Lang laufende Transaktionen sind daher in jeder MVCC-Datenbank ein betriebliches Warnsignal und ein Grund, analytische Abfragen von der operativen Datenbank fernzuhalten. Unter **Read Committed** mit MVCC erhält jede *Anweisung* einen frischen Snapshot; bei **Snapshot Isolation** verwendet die ganze *Transaktion* einen einzigen.

Echte Standardwerte und warum die Namen in die Irre führen

Namen von Isolationsebenen sind nicht portabel. Dasselbe Wort kann in verschiedenen Produkten unterschiedliche Garantien bedeuten, und die Standardwerte unterscheiden sich. Martin Kleppmanns Projekt Hermitage hat das tatsächliche Verhalten vieler Datenbanken gegen die obigen Anomalien getestet; die Herstellerdokumentation bestätigt die Standardwerte. Betrachten Sie diese Tabelle als Ausgangspunkt und prüfen Sie die Dokumentation der Version, die Sie betreiben.

Gleiche Namen, anderes Verhalten: was reale Datenbanken tun (zusammengefasst aus Herstellerdokumentation und den Hermitage-Tests)

Datenbank	Standardebene	Was die Ebenen wirklich sind	Anmerkungen
PostgreSQL	Read Committed	Read Uncommitted verhält sich wie Read Committed. Repeatable Read ist Snapshot Isolation. Serializable ist Serializable Snapshot Isolation (SSI).	Repeatable Read bricht eine Transaktion ab, die eine seit ihrem Snapshot geänderte Zeile ändern will (*could not serialize access due to concurrent update*), sodass Lost Updates verhindert werden; Write Skew nicht.
MySQL (InnoDB)	Repeatable Read	Einfache SELECTs lesen einen Snapshot, der beim ersten Lesen entsteht; sperrende Lesevorgänge, UPDATE und DELETE arbeiten mit Record- und Gap-Locks auf den neuesten festgeschriebenen Zeilen. Serializable macht einfache SELECTs zu sperrenden Lesevorgängen, wenn Autocommit aus ist.	Bei Repeatable Read kann ein Read-Modify-Write, geschrieben als SELECT und dann UPDATE, immer noch ein Update verlieren oder Write Skew erzeugen, sofern das SELECT nicht FOR UPDATE verwendet.
Oracle Database	Read Committed	Bietet Read Committed und Serializable; sein Serializable ist Snapshot Isolation.	Verhindert Lost Updates bei Serializable, lässt aber Write Skew zu.
Microsoft SQL Server	Read Committed (sperrend) auf SQL Server; Azure SQL Database schaltet standardmäßig Read Committed Snapshot ein	Sperrendes Read Committed, Read Committed Snapshot, Repeatable Read (sperrend), Snapshot (Snapshot Isolation) und Serializable (sperrend).	Auf SQL Server muss Snapshot Isolation vor der Nutzung pro Datenbank aktiviert werden (ALLOW_SNAPSHOT_ISOLATION).
CockroachDB	Serializable	Standardmäßig Serializable; Read Committed ist ebenfalls verfügbar.	Clients müssen Transaktionen wiederholen, die mit einem Serialisierungsfehler scheitern.

ACHTUNG

Zwei Fallen in den Namen

Erstens sind „Repeatable Read“ in PostgreSQL und in MySQL verschiedene Dinge: Das von PostgreSQL ist echte Snapshot Isolation, die konkurrierende Updates abbricht, das von MySQL mischt Snapshot-Lesevorgänge mit sperrenden Schreibvorgängen auf den neuesten Daten und kann Updates verlieren, die als Lesen-dann-Schreiben formuliert sind. Zweitens ist „Serializable“ in Oracle Snapshot Isolation und lässt Write Skew zu. Code, der auf einer Datenbank korrekt ist, kann auf einer anderen mit gleichnamiger Einstellung falsch sein.

Serialisierbarkeit erreichen

Datenbanken mit echter Serialisierbarkeit verwenden eine von drei Techniken. **Tatsächlich serielle Ausführung** lässt Transaktionen nacheinander auf einem einzigen Thread laufen, was schnell ist, wenn jede Transaktion kurz ist und die Daten im Arbeitsspeicher liegen (VoltDB sowie Redis-Skripte und -Transaktionen arbeiten so). **Two-Phase Locking (2PL)**, verwendet von den Serializable-Ebenen von SQL Server und MySQL, lässt Leser gemeinsame Sperren und Schreiber exklusive Sperren nehmen, einschließlich Bereichssperren gegen Phantome, und hält sie bis zum Ende der Transaktion; das ist korrekt, aber Leser und Schreiber blockieren einander, der Durchsatz sinkt bei Konkurrenz, und Deadlocks sind häufig. **Serializable Snapshot Isolation (SSI)**, 2008 von Michael Cahill, Uwe Röhm und Alan Fekete beschrieben und in PostgreSQL seit Version 9.1 verwendet (CockroachDB nutzt eine verwandte optimistische Technik), führt Transaktionen ohne Blockieren auf Snapshots aus, verfolgt Lese-Schreib-Abhängigkeiten zwischen ihnen und bricht eine Transaktion ab, wenn ein gefährliches Muster auftritt.

Welche Technik auch verwendet wird: Serialisierbare Transaktionen können mit einem Serialisierungsfehler (SQLSTATE `40001` in PostgreSQL und CockroachDB) oder einem Deadlock scheitern, und die Anwendung muss die ganze Transaktion wiederholen. Das sind die versteckten Kosten der stärksten Ebene: Jeder transaktionale Codepfad braucht eine Wiederholungsschleife, die gefahrlos wiederholbar ist, also keine Nebenwirkungen wie den Versand von E-Mails oder Aufrufe von Zahlungsanbietern innerhalb der Transaktion.

Connection Pooling

Jede Transaktion belegt eine Datenbankverbindung, und Verbindungen sind begrenzt (Abschnitt 9.1). Anwendungen halten daher einen **Pool** und leihen für jede Arbeitseinheit eine Verbindung aus. Größer ist nicht besser: Die Hinweise von HikariCP zitieren eine Oracle-Demonstration, in der die Reduzierung von 2048 Verbindungen auf 96 die Antwortzeiten von

etwa 100 ms auf etwa 2 ms senkte, und nennen (Kerne × 2) + effektive Festplatten als Ausgangswert zum Testen. Die Summe ist Poolgröße mal Instanzen, horizontales Skalieren kann die Datenbank also erschöpfen; ein zentraler Pooler wie PgBouncer hilft. Sein Transaktionsmodus verleiht eine Verbindung jeweils für eine Transaktion, doch Sitzungsfunktionen wie `SET` und Session Advisory Locks versagen dann. Halten Sie nie eine Verbindung, während Sie einen anderen Service aufrufen, sonst leert ein langsamer Zahlungsanbieter den Pool.

Pessimistische und optimistische Nebenläufigkeitskontrolle

In der Praxis laufen die meisten Anwendungen mit Read Committed oder Repeatable Read und schützen bestimmte Read-Modify-Write-Pfade explizit. Dafür gibt es drei Werkzeuge. Das beste, wenn möglich, ist ein **atomares Update**, bei dem die Datenbank Rechnung und Prüfung in einer Anweisung erledigt, etwa `UPDATE accounts SET balance = balance - 30 WHERE id = 7 AND balance ⩾ 30`, und bei dem man danach prüft, dass genau eine Zeile geändert wurde. Unter Read Committed in PostgreSQL lässt ein nebenläufiges Update die zweite Anweisung warten und ihre Bedingung dann gegen die neue Zeile neu auswerten, sodass kein Update verloren geht.

Braucht die Entscheidung Logik, die die Datenbank nicht ausdrücken kann, verwenden Sie pessimistisches Sperren: Sperren Sie die gelesenen Zeilen mit `SELECT ... FOR UPDATE`, sodass nebenläufige Transaktionen, die sie brauchen, bis zu Ihrem Commit warten. Das ist einfach und passt zu kurzen, stark umkämpften Operationen (einen Kurier einer Bestellung zuweisen). Die Kosten sind Wartezeit, das Risiko von Deadlocks (sperren Sie Zeilen in einer festen Reihenfolge, etwa nach aufsteigender id) und geringerer Durchsatz. Varianten sind `NOWAIT`, das sofort scheitert, statt zu warten, und `SKIP LOCKED`, das bereits gesperrte Zeilen überspringt und der übliche Weg ist, eine zuverlässige Job-Queue auf einer relationalen Tabelle zu bauen.

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

Pessimistisches Sperren: Die zweite Abhebung wartet beim SELECT, bis die erste festgeschrieben ist, und liest dann den neuen Saldo.

Optimistische Nebenläufigkeitskontrolle nimmt keine Sperren, während der Nutzer oder der Code nachdenkt. Jede Zeile trägt eine Versionsnummer (oder einen Zeitstempel oder Hash). Der Schreiber ändert nur, wenn die Version noch die gelesene ist, und erhöht sie. Ändert sich keine Zeile, war jemand anderes schneller, und die Anwendung lädt neu und wiederholt oder informiert den Nutzer. Das passt zu Daten mit wenig Konkurrenz und langer „Denkzeit“: Eine Restaurantleiterin, die fünf Minuten lang im Browser eine Speisekarte bearbeitet, darf nicht so lange eine Datenbanksperre halten. HTTP hat dieselbe Idee eingebaut, mit ETags und dem Header `If-Match`.

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

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

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

Optimistische Nebenläufigkeit mit einer Versionsspalte: Das Update gelingt nur, wenn seit dem Lesen niemand den Speisekartenartikel geändert hat.

AUSGEARBEITETES BEISPIEL

Ausgearbeitetes Beispiel: Write Skew mit zwei Ärzten in Bereitschaft

Ein Dienstplansystem in einem Krankenhaus setzt eine Regel durch: Für jede Schicht muss mindestens ein Arzt Bereitschaft haben. Alice und Bob haben heute Nacht beide Bereitschaft. Beide fühlen sich unwohl und drücken fast im selben Moment auf „Bereitschaft beenden“. Jede Anfrage führt diese Logik in einer Transaktion aus: Zähle die Ärzte in Bereitschaft für die Schicht; sind es mindestens zwei, setze meine eigene Zeile auf „keine Bereitschaft“. Die Datenbank läuft mit Snapshot Isolation (Repeatable Read in PostgreSQL oder Serializable in Oracle).

Start	Tabelle `doctors`: Alice on_call = true, Bob on_call = true. Regel: count(on_call) >= 1.
Schritt 1: T1 (Alice) beginnt	T1 nimmt einen Snapshot.
Schritt 2: T2 (Bob) beginnt	T2 nimmt einen Snapshot. Beide Snapshots zeigen zwei Ärzte in Bereitschaft.
Schritt 3: T1 liest	`SELECT count(*) FROM doctors WHERE shift = 1234 AND on_call` liefert 2. Prüfung bestanden.
Schritt 4: T2 liest	Dieselbe Abfrage liefert auch im Snapshot von T2 den Wert 2. Prüfung bestanden.
Schritt 5: T1 schreibt	`UPDATE doctors SET on_call = false WHERE name = 'Alice' AND shift = 1234`
Schritt 6: T2 schreibt	`UPDATE doctors SET on_call = false WHERE name = 'Bob' AND shift = 1234`. Eine andere Zeile, daher wird kein Schreib-Schreib-Konflikt erkannt.
Schritt 7: beide schreiben fest	Snapshot Isolation sieht keinen Konflikt: Jede Transaktion hat eine Zeile geschrieben, die die andere nicht geschrieben hat. Ergebnis: Niemand hat Bereitschaft. Jede Transaktion war für sich korrekt; gemeinsam haben sie die Regel verletzt.
Unter Serializable (SSI)	Die Datenbank bemerkt, dass jede Transaktion Daten gelesen hat, die die andere danach geändert hat (ein gefährlicher Zyklus von Lese-Schreib-Abhängigkeiten). Ein Commit scheitert mit SQLSTATE 40001; die Wiederholung zählt neu, findet einen Arzt und lehnt ab.

Lösung auf einer schwächeren Ebene	Sperren Sie die Zeilen, von denen die Entscheidung abhängt: `SELECT name FROM doctors WHERE shift = 1234 AND on_call FOR UPDATE`, und zählen Sie dann die zurückgegebenen Zeilen. Jetzt wartet T2 auf die Sperren von T1. Unter Read Committed liest T2 danach erneut, findet nur Bob in Bereitschaft und lehnt ab; unter Repeatable Read in PostgreSQL scheitert es stattdessen mit einem Serialisierungsfehler, und die Wiederholung lehnt ab.

Write Skew ist eine Verallgemeinerung des Lost Update: Die Transaktionen lesen dieselben Daten, schreiben aber *verschiedene* Zeilen, sodass die First-Committer-Wins-Regel der Snapshot Isolation nie greift.

Dasselbe Muster tritt in Buchungssystemen auf (zwei Personen buchen den letzten Platz oder überlappende Besprechungstermine), bei Benutzernamen (zwei Registrierungen prüfen, ob ein Name frei ist) und im Bankwesen (zwei Abhebungen von verschiedenen Konten, die zusammen ein gemeinsames Überziehungslimit überschreiten).

Wenn die Zeilen, die einen Konflikt belegen würden, noch nicht existieren, wie bei zwei überlappenden Buchungen, hat `FOR UPDATE` nichts zu sperren. Möglich sind dann Serializable, ein Unique- oder Exclusion-Constraint (PostgreSQL kann überlappende Zeiträume deklarativ verbieten) oder das *Materialisieren des Konflikts*: Man legt Zeilen an, etwa eine pro Raum und Zeitfenster, nur damit sie sich sperren lassen.

WENN DAS NEU FÜR SIE IST

Schritt für Schritt: ein Lesen-dann-Schreiben sicher machen

Der Punkt, an dem hier die meisten stolpern, ist, dass ein Lese- und ein Schreibvorgang in derselben Transaktion nicht verhindern, dass sich eine andere Transaktion dazwischenschiebt. Zwei Bestellungen für ein Gericht zeigen, warum, und wie man es behebt.

- Das Tagesgericht eines Restaurants hat noch 8 Portionen, gespeichert in einer Zeile. Die Bestellungen X (2 Portionen) und Y (3) kommen gleichzeitig an, beide unter Read Committed, der Voreinstellung in PostgreSQL und Oracle.
- Jede liest 8, zieht ihre Portionen ab und speichert: X speichert 6, dann speichert Y 5. Die 2 Portionen von X sind aus der Zählung verschwunden, ein Lost Update, sodass noch 2 Portionen verkauft werden könnten, die es nicht gibt. Atomarität hat nicht geholfen: Sie betrifft Fehler, nicht das Abwechseln.
- Lösung eins, ein atomares Update: Die Datenbank erledigt Rechnung und Prüfung in einer Anweisung, etwa „nimm 3, wenn mindestens 3 übrig sind“. In PostgreSQL wartet die Anweisung von Y auf den Commit von X, prüft erneut gegen 6 und lässt 3 übrig.
- Lösung zwei, eine pessimistische Sperre: Jede Transaktion sperrt die Zeile beim Lesen. X ist zuerst da, also wartet der Lesevorgang von Y, bis X festgeschrieben hat, und sieht dann 6, nicht 8.
- Lösung drei, optimistische Nebenläufigkeit: keine Sperre, aber jeder Schreibvorgang nennt die gelesene Versionsnummer und gelingt nur, wenn die Zeile diese Nummer noch trägt. Der Schreibvorgang von X geht durch und erhöht die Version; der von Y trifft keine Zeile, also liest Y erneut und versucht es mit 6 noch einmal.
- Serializable verhindert es ebenfalls, kann aber eine Transaktion scheitern lassen, die dann wiederholt werden muss.
- **Die Lehre:** Die Gefahr steckt in jedem Lesen-Entscheiden-Schreiben, wie auch immer die Isolationsebene heißt. Schützen Sie jedes davon mit einer atomaren Anweisung, einer Sperre, einer Versionsprüfung oder Serializable mit Wiederholungen.

Eine Isolationsstrategie wählen

Es gibt nicht die eine richtige Ebene; es gibt eine richtige Ebene für jedes Stück Logik. Ein praktischer Ansatz geht von den Invarianten aus, nicht von den Einstellungen.

Checkliste: einen transaktionalen Codepfad sicher machen

- [ ] **Die Invariante benennen**
  Schreiben Sie die Regel auf, die der Code schützt (Saldo nie unter null, ein Kurier pro Bestellung, mindestens ein Arzt in Bereitschaft). Gibt es keine, genügt wahrscheinlich die Standardebene.
- [ ] **Constraints und atomare Anweisungen bevorzugen**
  Eindeutige Indizes, Fremdschlüssel, CHECK- und Exclusion-Constraints sowie bedingte Updates in einer einzigen Anweisung werden von der Datenbank unabhängig von der Isolationsebene durchgesetzt.
- [ ] **Read-Modify-Write erkennen**
  Jeder Code, der einen Wert liest, in der Anwendung entscheidet und zurückschreibt, birgt das Risiko eines Lost Update oder Write Skew. Schützen Sie ihn mit FOR UPDATE, einer Versionsspalte oder Serializable.
- [ ] **Die echte Semantik Ihrer Datenbank kennen**
  Prüfen Sie, was die Ebenennamen Ihres Produkts tatsächlich bedeuten (Repeatable Read in PostgreSQL und MySQL unterscheiden sich; Serializable in Oracle ist Snapshot Isolation).
- [ ] **Richtig wiederholen**
  Serializable und Repeatable Read können mit Serialisierungsfehlern und Deadlocks abbrechen. Wiederholen Sie die ganze Transaktion mit Backoff, und halten Sie externe Nebenwirkungen außerhalb davon.
- [ ] **Transaktionen kurz halten**
  Keine Denkzeit von Nutzern, keine Netzwerkaufrufe und keine großen Berichte innerhalb einer Transaktion; lange Transaktionen halten Sperren und blähen den MVCC-Speicher auf.
- [ ] **Unter Nebenläufigkeit testen**
  Führen Sie die kritischen Pfade in einem Integrationstest nebenläufig gegen die echte Datenbank-Engine aus, nicht gegen einen In-Memory-Ersatz mit anderer Semantik.

Als Faustregel ist Read Committed mit expliziten Sperren oder Versionsprüfungen für die meisten Geschäftsanwendungen eine solide, schnelle Voreinstellung, sofern das Team diszipliniert nach Read-Modify-Write-Pfaden sucht. Serializable ist die sicherere Voreinstellung, wenn die Regeln komplex sind, das Team groß ist oder eine verletzte Invariante teuer wird (Geld, Medizin, Betriebssicherheit), und viele Teams setzen es inzwischen für die kritischen Pfade ein und nehmen Wiederholungen in Kauf. Hier sind sich Fachleute wirklich

uneinig: Manche argumentieren, dass Serializable als Voreinstellung eine ganze Klasse von Fehlern verhindert, die kein Code-Review je findet; andere argumentieren, dass seine Wiederholungen und seine Konkurrenz um Ressourcen zu viel kosten und dass explizite Sperren die Absicht sichtbar machen. Beide Positionen sind vertretbar, wenn die Wahl bewusst getroffen und getestet ist.

PRÜFEN SIE IHR VERSTÄNDNIS

▼ **Warum ist Snapshot Isolation nicht serialisierbar?**

Sie lässt Write Skew zu: Zwei Transaktionen können überlappende Daten lesen und verschiedene Zeilen schreiben und so gemeinsam eine Regel verletzen, die jede geprüft hat, ohne dass ein Schreib-Schreib-Konflikt erkannt wird.

▼ **Wovon hängt das C in ACID ab?**

Hauptsächlich von der Anwendung: Die Datenbank setzt deklarierte Constraints durch, aber Geschäftsinvarianten gelten nur, wenn die Transaktionen der Anwendung sie wahren.

▼ **Wann würden Sie optimistische statt pessimistischer Nebenläufigkeitskontrolle wählen?**

Wenn Konflikte selten sind oder zwischen Lesen und Schreiben eine lange Denkzeit liegt, etwa wenn ein Nutzer ein Formular bearbeitet, sodass das Halten von Sperren verschwenderisch oder unmöglich wäre.

▼ **Warum muss Code, der Serializable verwendet, gefahrlos wiederholbar sein?**

Serialisierbare Transaktionen können durch Serialisierungsfehler oder Deadlocks abgebrochen werden, und die richtige Reaktion ist, die ganze Transaktion erneut auszuführen; daher darf sie keine externen Nebenwirkungen haben.

▼ **Probieren Sie es aus: Zwei Fans kaufen das letzte Ticket. Jede Transaktion liest remaining = 1 und schreibt dann bei Read Committed remaining = 0, sodass beide erfolgreich sind. Benennen Sie das Problem, und nennen Sie die einfachste Lösung, die Sie verwenden würden, sowie eine Alternative.**

Das ist ein Lost Update: Beide haben gelesen, in Anwendungscode entschieden und zurückgeschrieben. Die einfachste Lösung ist eine atomare bedingte Aktualisierung: UPDATE shows SET remaining = remaining - 1 WHERE id = ? AND remaining > 0, dann prüfen, ob sich eine Zeile geändert hat; wenn nicht, ist die Vorstellung ausverkauft. Alternativ verwenden Sie SELECT ... FOR UPDATE oder eine Versionsspalte mit Wiederholung. Ein CHECK-Constraint, dass remaining mindestens 0 ist, bietet ein letztes Sicherheitsnetz.

QUELLEN

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

Replikation, Partitionierung und Sharding

BEVOR SIE DAS LESEN

Dieser Abschnitt baut auf:

- 0.5 Daten speichern
- 2.5 Verfügbarkeit und Zuverlässigkeit
- 6.6 CAP und Konsistenz

Begriffe von früher, die er verwendet: Verfügbarkeit, Latenz, Index, Write-Ahead-Log (WAL), Linearisierbarkeit

DIE IDEE IN KÜRZE

Stellen Sie sich eine beliebte Bibliothek vor. Um weiter Leser bedienen zu können, wenn ein Gebäude überflutet wird, hält sie identische Kopien ihres Katalogs in mehreren Zweigstellen: Das ist Replikation. Die Kopien werden per Kurier aktualisiert, daher kann eine Zweigstelle etwas hinterherhinken, und wer gerade in einer Zweigstelle ein Buch zurückgegeben hat, findet es in einer anderen womöglich noch als ausgeliehen verzeichnet. Wird der Bestand zu groß für ein einzelnes Gebäude, teilt die Bibliothek ihn auf, A bis M hier und N bis Z dort: Das ist Partitionierung. Gut aufteilen heißt, dass keine Zweigstelle überlastet ist und Leser selten jede Zweigstelle aufsuchen müssen, um eine Antwort zu erhalten.

WAS SIE WISSEN MÜSSEN

- Replikation hält Kopien derselben Daten auf mehreren Knoten für Verfügbarkeit, Leseskalierung und Lokalität; Partitionierung (Sharding) verteilt Daten über Knoten für Schreibdurchsatz und Speicherplatz; reale Systeme kombinieren meist beides.
- Bei der Leader-Follower-Replikation gehen alle Schreibvorgänge an einen Leader; synchrone Follower schützen bestätigte Schreibvorgänge auf Kosten der Latenz, asynchrone Follower sind schnell, können bei einem Failover aber jüngste Schreibvorgänge verlieren.
- Replikationsverzögerung bricht Read-your-Writes, Monotonic Reads und Consistent Prefix Reads; Abhilfe schaffen etwa das Lesen eigener jüngster Schreibvorgänge vom Leader, das Warten auf eine Log-Position und das feste Zuordnen von Nutzern zu einem Replikat.
- Ein Failover muss den Ausfall erkennen, den aktuellsten Knoten wählen und den alten Leader abschotten (Fencing); Split Brain, bei dem zwei Knoten Schreibvorgänge annehmen, kann lange, schmerzhafte Wiederherstellungen verursachen, wie GitHubs

Vorfall von 2018 zeigte.

- Multi-Leader-Replikation muss Schreibkonflikte auflösen; Leaderless Replication (im Stil von Dynamo) nutzt Quoren mit R + W > N, Read Repair und Anti-Entropy, aber Quoren allein garantieren keine Linearisierbarkeit.
- Bereichspartitionierung unterstützt Bereichsscans, erzeugt aber Hotspots bei zeitlich geordneten Schlüsseln; Hash-Partitionierung verteilt die Last, zerstreut aber Bereiche; Konsistentes Hashing oder feste logische Partitionen halten das Rebalancing günstig.
- Hot Keys, Schieflage und Sekundärindizes (lokal heißt Scatter-Gather-Lesevorgänge; global heißt entfernte, meist asynchrone Indexschreibvorgänge) müssen von den wichtigsten Abfragen her entworfen werden.

Zwei verschiedene Gründe, Daten zu verteilen

Ein einzelner Datenbankserver stößt irgendwann an eine Grenze: Verfügbarkeit (er ist eine Maschine, die ausfallen kann), Lesekapazität, Schreibkapazität, Speicherplatz oder Nähe zu Nutzern auf anderen Kontinenten. Es gibt zwei einander ergänzende Wege, Daten über mehrere Maschinen zu verteilen, und es ist wichtig, sie nicht zu verwechseln. Replikation hält eine *Kopie derselben Daten* auf mehreren Knoten, für Verfügbarkeit, Leseskalierung und geringere Latenz in der Nähe der Nutzer. Partitionierung, in vielen Produkten *Sharding* genannt, *teilt die Daten auf*, sodass jeder Knoten eine Teilmenge hält, für Schreibdurchsatz und Speicherplatz über eine Maschine hinaus. Reale Systeme tun meist beides: Jede Partition wird repliziert.

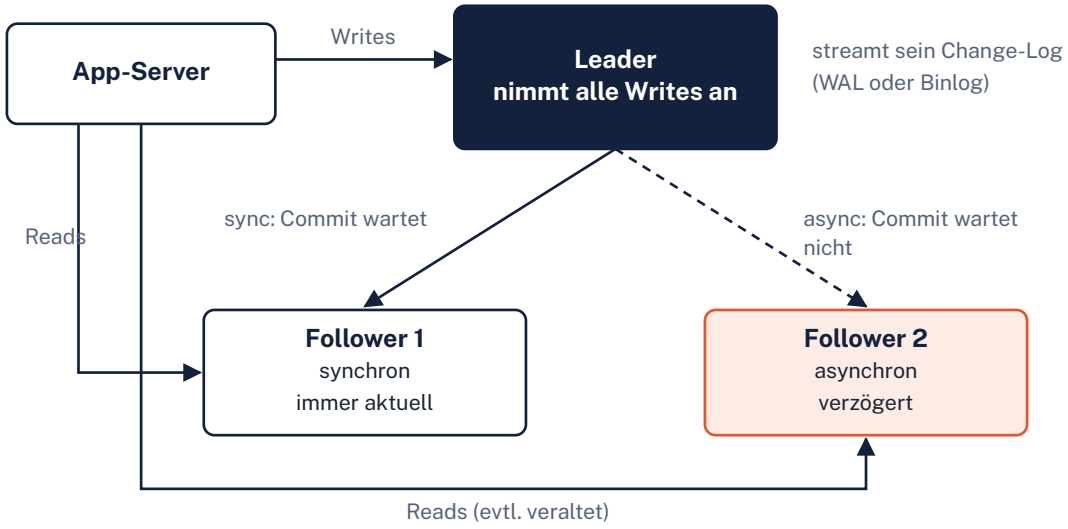
Beide haben ernste Folgen für die Korrektheit. Replikation wirft die Frage auf, was ein Leser sieht, solange die Kopien nicht übereinstimmen; das ist das Thema von Konsistenzmodellen, CAP und PACELC. Partitionierung wirft die Frage auf, was mit Abfragen, Indizes und Transaktionen geschieht, die mehrere Partitionen umfassen. Dieser Abschnitt erklärt die Mechanismen so gründlich, dass Sie die Aussagen einer Datenbank beurteilen und einen Shard-Schlüssel entwerfen können.

Leader-Follower-Replikation

Das häufigste Schema ist die Leader-Follower-Replikation (auch Primary-Replica genannt; ältere Dokumentation sagt Master-Slave). Ein Knoten, der Leader, nimmt alle Schreibvorgänge an. Er zeichnet jede Änderung in seinem Log auf (dem Write-Ahead-Log von PostgreSQL, dem Binary Log von MySQL) und streamt dieses Log an die Follower, die die Änderungen in derselben Reihenfolge anwenden. Lesevorgänge können an den Leader oder an Follower gehen. PostgreSQL Streaming Replication, MySQL-Replikation, SQL Server Availability Groups, MongoDB Replica Sets und die meisten verwalteten Cloud-Datenbanken arbeiten so.

Die zentrale Entwurfsentscheidung ist, wann der Leader einen Commit als erfolgreich meldet. Bei **synchroner** Replikation wartet er, bis ein Follower bestätigt, dass er die Änderung hat, sodass ein festgeschriebener Schreibvorgang den Verlust des Leaders übersteht; der Preis ist, dass jeder Commit einen Netzwerk-Roundtrip enthält und Schreibvorgänge stocken, wenn der synchrone Follower ausgefallen ist. Bei **asynchroner** Replikation bestätigt der Leader sofort und versendet die Änderung danach; Commits sind schnell, und Follower können weit entfernt sein, aber wenn der Leader ausfällt, gehen die letzten Änderungen verloren, die noch keinen Follower erreicht hatten, obwohl dem Client gemeldet wurde, sie seien festgeschrieben. Ein verbreiteter Kompromiss, in MySQL semisynchron genannt, wartet auf einen Follower und lässt die übrigen asynchron laufen. In der Bank ist der Verlust einer bestätigten Zahlung inakzeptabel, daher ist mindestens eine synchrone Kopie (oder eine konsensbasierte Datenbank) üblich; für die Speisekarten-Datenbank der Essenslieferplattform kann ein möglicher Verlust von wenigen Sekunden ein akzeptabler Preis für Geschwindigkeit sein. Das ist die RPO-Abwägung aus Verfügbarkeit und Zuverlässigkeit in konkreter Form.

Leader mit einem synchronen und einem asynchronen Follower

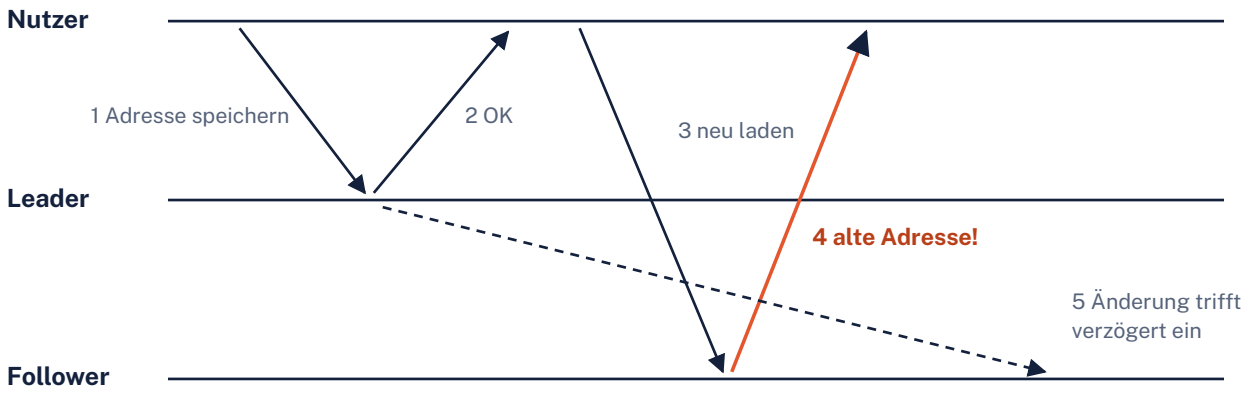


Alle Schreibvorgänge gehen an den Leader; sein Change-Log fließt zu den Followern, und der Commit wartet nur auf den synchronen.

Replikationsverzögerung und die Anomalien, die sie verursacht

Asynchrone Follower liegen meist Millisekunden zurück, aber unter Last, während eines großen Batch-Updates oder über ein langsames Netzwerk kann die Replikationsverzögerung auf Sekunden oder Minuten anwachsen. Lesen von Followern erzeugt dann Anomalien, die Nutzer bemerken. Douglas Terry und Kollegen definierten die ersten beiden Garantien unten, Read-your-Writes und Monotonic Reads, 1994 als *Session Guarantees*; die dritte ist vor allem wichtig, wenn die Daten zusätzlich partitioniert sind.

Read-your-Writes verletzt



Lösung: eigene jüngste Änderungen vom Leader lesen oder warten, bis der Follower aufholt

Die eigene Änderung der Nutzerin ist vom Leader bestätigt, aber noch nicht auf dem Follower angewendet, der ihren nächsten Lesevorgang bedient.

Verzögerungsanomalien und ihre Lösungen

Garantie	Was ohne sie schiefgeht	Übliche Lösungen
Read-your-Writes (Read-after-Write)	Eine Kundin ändert ihre Lieferadresse, lädt neu und sieht die alte: Es sieht aus, als sei das Speichern fehlgeschlagen, also versucht sie es erneut.	Daten, die der Nutzer gerade geändert haben könnte, vom Leader lesen (zum Beispiel eine Minute lang nach seinem letzten Schreibvorgang); oder beim Schreiben die Log-Position des Leaders festhalten und erst von einem Follower lesen, wenn er diese Position angewendet hat.
Monotonic Reads	Die App eines Kuriers aktualisiert zweimal; der erste Lesevorgang trifft einen aktuellen Follower und zeigt eine Bestellung als zugewiesen, der zweite trifft einen verzögerten, und die Bestellung scheint wieder nicht zugewiesen. Die Zeit scheint rückwärts zu laufen.	Die Lesevorgänge jedes Nutzers an denselben Follower leiten (zum Beispiel über einen Hash der Nutzer-ID) und nur dann auf einen anderen ausweichen, wenn er nicht verfügbar ist.
Consistent Prefix Reads	Leser sehen eine Antwort vor der Frage oder das Ereignis „zugestellt“ einer Bestellung vor „abgeholt“, weil beide in verschiedene Partitionen geschrieben wurden, die unterschiedlich verzögert sind.	Kausal zusammenhängende Daten in dieselbe Partition schreiben oder kausale Abhängigkeiten explizit verfolgen.

Keine dieser Anomalien tritt auf, wenn jeder Lesevorgang an den Leader geht; die einfachste Lösung ist daher oft, Lesevorgänge, die frisch sein müssen, an den Leader zu senden und Follower nur für Lesevorgänge zu nutzen, die veraltete Daten vertragen: Produktlisten, Berichte, Suchfeeds. Entscheiden Sie das bewusst pro Abfrage, statt standardmäßig „alle Lesevorgänge an Replikate“ zu leiten.

Failover und Split Brain

Failover ersetzt einen ausgefallenen Leader durch einen Follower. Es hat drei Schritte: erkennen, dass der Leader ausgefallen ist (meist über ein Timeout), den neuen Leader wählen (idealerweise den aktuellsten Follower, oft über ein Konsensprotokoll wie Raft zwischen den Knoten oder einen externen Koordinator wie etcd oder ZooKeeper) und die Schreibvorgänge auf ihn umleiten. Jeder Schritt kann schiefgehen. Ein zu kurzes Timeout löst bei einem kurzen Netzwerkaussetzer oder einer Lastspitze ein Failover aus, was die Lage verschlimmert; ein zu langes verlängert den Ausfall. Bei asynchroner Replikation gehen Schreibvorgänge verloren, die der alte Leader noch nicht versendet hatte, und kommt der alte Leader zurück und hält sie noch, geraten sie mit neuen Schreibvorgängen in Konflikt.

Der gefährlichste Ausfall ist Split Brain: Zwei Knoten halten sich beide für den Leader und nehmen beide Schreibvorgänge an, sodass die Daten auf eine Weise auseinanderlaufen, die sich womöglich nicht automatisch zusammenführen lässt. Es folgt typischerweise auf eine Netzwerkpartition, bei der der alte Leader abgeschnitten, aber noch aktiv ist. Schutz erfordert ein Quorum (ein Knoten darf nur führen, solange eine Mehrheit zustimmt), *Fencing* (die Autorität des neuen Leaders wird durch ein ständig steigendes Token durchgesetzt, das der Speicher prüft, oder der alte Leader wird zwangsweise abgeschaltet, eine Technik mit dem Spitznamen STONITH: shoot the other node in the head) und Clients, die zuverlässig dem aktuellen Leader folgen.

GitHubs Vorfall vom 21. Oktober 2018 zeigt, wie schwierig das in der Praxis ist. Ein Netzwerkfehler trennte 43 Sekunden lang die Verbindung zwischen seinem Netzknoten an der US-Ostküste und seinem Hauptrechenzentrum an der Ostküste. Seine Failover-Werkzeuge beförderten, wie vorgesehen, Datenbank-Primaries an der Westküste. Als die Verbindung zurückkehrte, hielten beide Küsten Schreibvorgänge, die der anderen fehlten, sodass GitHub nicht einfach zurückschalten konnte; zum Schutz der Datenintegrität entschied es sich für Wiederherstellen und Abgleichen, und der Dienst war 24 Stunden und 11 Minuten lang eingeschränkt. Automatisches Failover über Regionen hinweg hatte aus einem Aussetzer von 43 Sekunden eine tagelange Wiederherstellung gemacht.

Backups und der Nachweis, dass sie sich wiederherstellen lassen

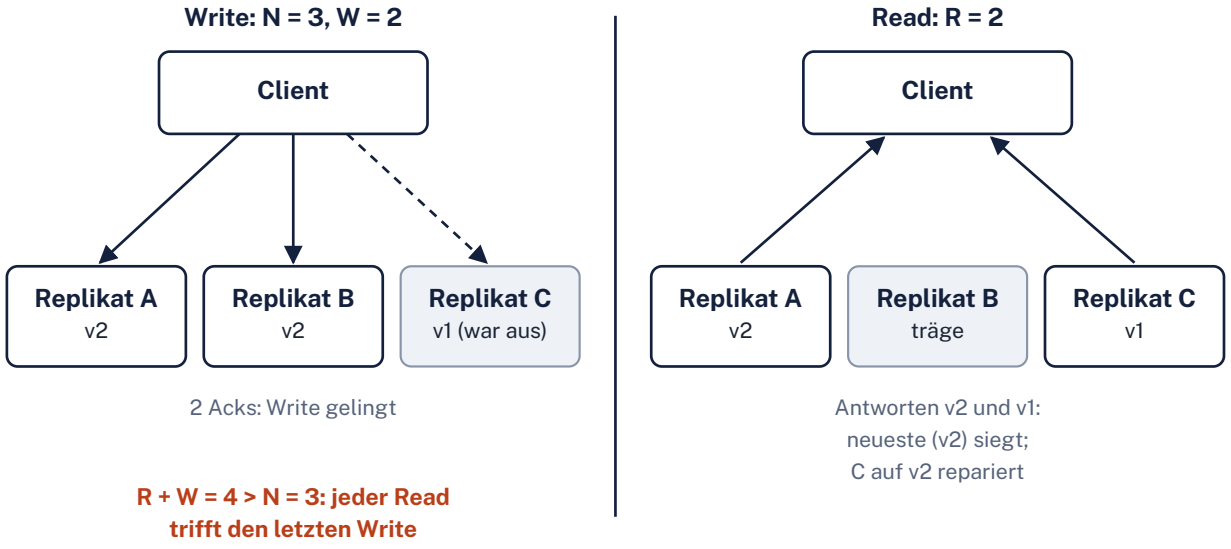
Replikation ist kein Backup: Eine Replik kopiert ein versehentliches `DELETE` getreulich mit. Das Base Backup von PostgreSQL plus kontinuierliches WAL-Archiv erlaubt die Wiederherstellung zu einem gewählten Zeitpunkt kurz vor dem Fehler, und Kopien sollten dort liegen, wo ein einzelner Vorfall nicht beide erreicht. Ein Backup zählt erst, wenn es wiederhergestellt wurde. GitLab lernte das im Januar 2017: Nachdem Produktionsdaten versehentlich gelöscht worden waren, stellte sich heraus, dass die regelmäßigen `pg_dump`-Backups wegen einer Versionsabweichung fehlgeschlagen waren, die Fehler-E-Mails abgewiesen worden waren und der Storage-Bucket leer war. Die Wiederherstellung gelang aus einem manuellen, etwa sechs Stunden alten Snapshot. Automatisieren Sie also regelmäßige Wiederherstellungen in eine isolierte Umgebung, prüfen Sie die Daten, messen Sie die benötigte Zeit an RTO und RPO und lösen Sie einen Alarm aus, wenn ein Backup oder ein Wiederherstellungstest fehlschlägt oder gar nicht läuft.

Multi-Leader- und Leaderless Replication

Bei der Multi-Leader-Replikation nehmen mehrere Knoten Schreibvorgänge an und replizieren untereinander. Sie wird für mehrere Rechenzentren genutzt (jede Region schreibt lokal, sodass Nutzer keine Latenz über Ozeane hinweg haben), für Clients, die offline arbeiten (eine Kalender- oder Notiz-App ist faktisch ein Leader auf jedem Gerät), und für kollaboratives Bearbeiten. Das unvermeidliche Problem sind **Schreibkonflikte**: Zwei Leader ändern denselben Datensatz gleichzeitig. Lösungen sind Last Write Wins (einfach, verwirft aber stillschweigend Daten und hängt von Uhren ab), das Zusammenführen von Werten, das Behalten beider Versionen mit Rückfrage bei einem Menschen oder Datentypen, die sich automatisch zusammenführen lassen (CRDTs, Conflict-free Replicated Data Types). Am zuverlässigsten ist es, Konflikte zu vermeiden: Geben Sie jedem Datensatz eine Heimatregion, die allein seine Schreibvorgänge annimmt, etwa das Heimatland eines Kunden.

Leaderless Replication, bekannt geworden durch Amazons Dynamo-Paper (DeCandia und Kollegen, 2007) und verwendet in Apache Cassandra, ScyllaDB und Riak, hat überhaupt keinen Leader. Der Client oder ein Koordinatorknoten in seinem Auftrag sendet jeden Schreibvorgang an alle N Replikate eines Schlüssels und wertet ihn als erfolgreich, wenn W von ihnen bestätigen; jeden Lesevorgang sendet er an Replikate, wartet auf R Antworten und nimmt die neueste Version. Gilt R + W > N, überschneidet sich jede Lesemenge mit jeder erfolgreichen Schreibmenge in mindestens einem Replikat, sodass ein Lesevorgang normalerweise den neuesten bestätigten Schreibvorgang sieht. Replikate, die Schreibvorgänge verpasst haben, werden durch *Read Repair* (der Koordinator schreibt die neueste Version zurück, wenn er eine veraltete sieht) und eine im Hintergrund laufende *Anti-Entropy* repariert, die Replikate vergleicht. Cassandra bietet diese Wahl pro Anfrage als Konsistenzstufen wie ONE, QUORUM und ALL an.

Ein Quorum-Schreib- und -Lesevorgang mit N = 3



Mit W = 2 und R = 2 hat mindestens ein Replikat in jedem Lesevorgang den neuesten bestätigten Schreibvorgang gesehen.

Quoren wird oft zu viel zugetraut. R + W > N macht das System nicht schon allein linearisierbar: Nebenläufige Schreibvorgänge, Schreibvorgänge, die auf manchen Replikaten scheiterten und auf anderen gelangen, *Sloppy Quorums* (Schreiben auf Ersatzknoten während Ausfällen) und Last Write Wins mit abweichenden Uhren lassen alle veraltete oder verlorene Daten zu. Kapitel 6 geht die Quorum-Arithmetik und diese Grenzfälle in CAP und PACELC durch. Leaderless-Speicher eignen sich hervorragend für hohe Schreibverfügbarkeit über Ausfälle und Rechenzentren hinweg; für Invarianten wie „denselben Platz nie zweimal verkaufen“ passen sie schlecht, sofern sie für solche Operationen keine Lightweight Transactions oder Konsensfunktion anbieten.

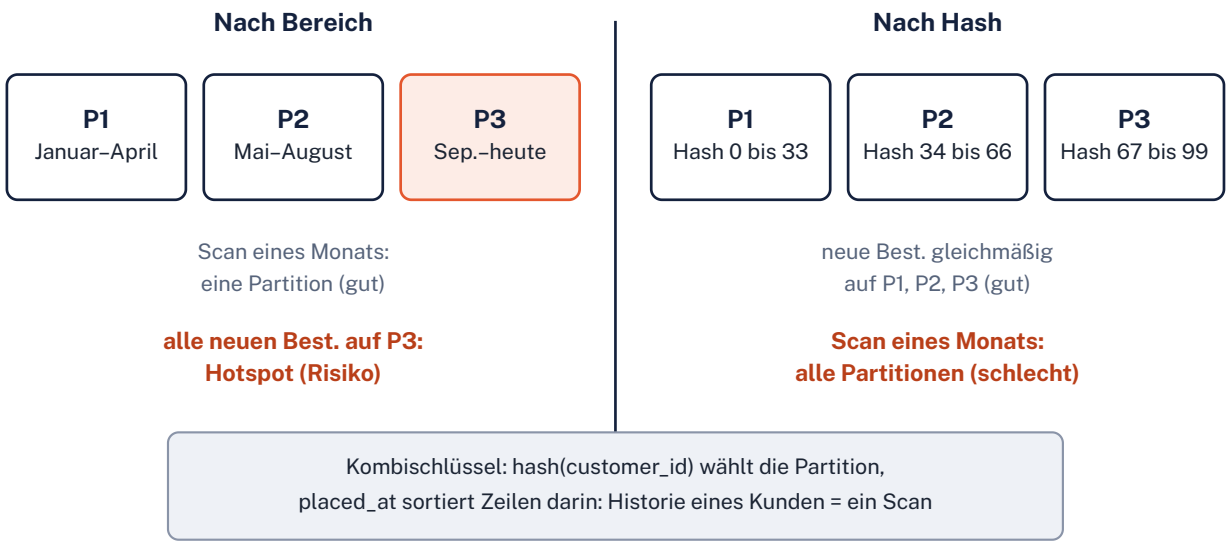
Partitionierung: nach Schlüsselbereich oder nach Hash

Partitionierung ordnet jeden Datensatz über seinen **Partitionsschlüssel** (oder Shard-Schlüssel) einer Partition zu, und das Ziel ist, Daten und Last gleichmäßig zu verteilen. Eine Partition, die unverhältnismäßig viel Last erhält, ist ein Hotspot, und eine ungleichmäßige Verteilung nennt man *Schieflage* (Skew). Es gibt zwei Grundschemata.

Schlüsselbereichspartitionierung gibt jeder Partition einen zusammenhängenden Bereich von Schlüsseln, wie die Bände eines Lexikons. Bereichsscans sind effizient, weil benachbarte Schlüssel zusammen gespeichert sind (alle Bestellungen vom Oktober in einer oder zwei Partitionen), und Partitionen lassen sich teilen, wenn sie wachsen, wie in Bigtable, HBase, Spanner und beim Range-Sharding von MongoDB. Die Gefahr ist ein Schlüssel, der mit der Zeit wächst: Ist der Schlüssel ein Zeitstempel oder eine fortlaufende Bestellnummer, landet jeder

neue Schreibvorgang in der letzten Partition, die zum Hotspot wird, während die anderen untätig sind. **Hash-Partitionierung** wendet eine Hashfunktion auf den Schlüssel an und weist Partitionen Bereiche von Hashwerten zu, sodass sich selbst fortlaufende Schlüssel gleichmäßig verteilen. Der Preis ist, dass Bereichsabfragen auf den Schlüssel nun jede Partition fragen müssen. Viele Speicher kombinieren beides mit einem *zusammengesetzten Schlüssel*: Cassandra hasht den Partitionsschlüssel, um den Knoten zu wählen, und sortiert die Zeilen innerhalb der Partition nach Clustering-Spalten, sodass `(customer_id, placed_at)` die Bestellungen eines Kunden zusammen und in zeitlicher Reihenfolge auf einem Knoten ablegt.

Bereichs- gegenüber Hash-Partitionierung für Bestellungen



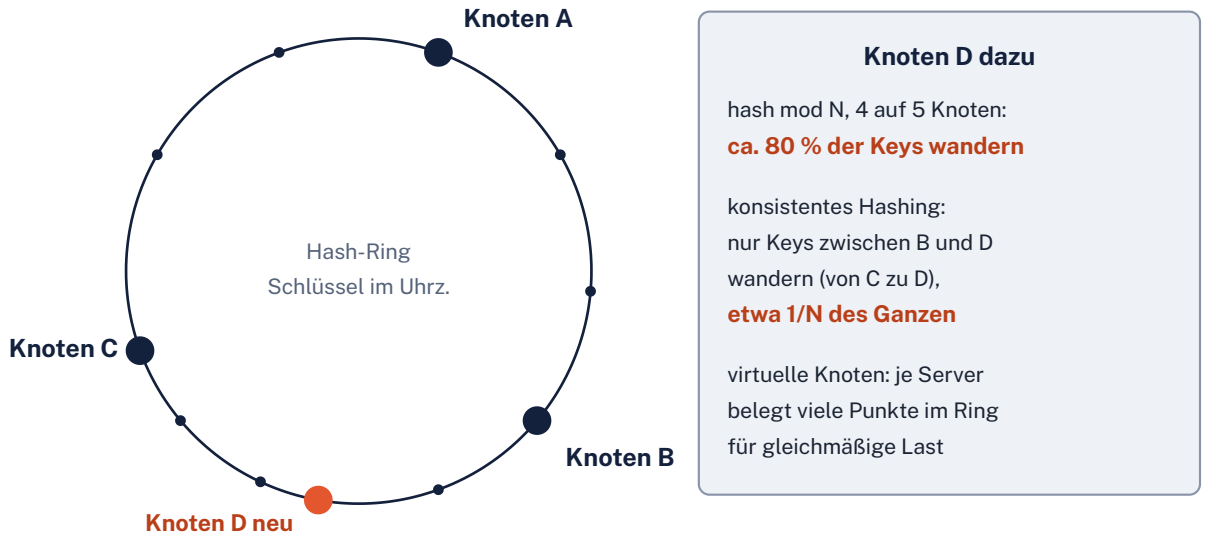
Bereichspartitionierung hält Bereiche zusammen, ballt aber zeitlich geordnete Schreibvorgänge; Hashing verteilt Schreibvorgänge, zerstreut aber Bereiche.

Konsistentes Hashing und Rebalancing

Der naheliegende Weg, Hashes Knoten zuzuordnen, `hash(key) mod N`, hat einen fatalen Fehler: Ändert sich N, wandert fast jeder Schlüssel. Der Schritt von 4 auf 5 Knoten verlagert etwa 80 % der Schlüssel (jeden Schlüssel, dessen Rest für 4 und 5 verschieden ist), der von 10 auf 11 etwa 91 %, was bedeutet, fast den ganzen Datenbestand zu kopieren, nur um eine Maschine hinzuzufügen. Konsistentes Hashing, 1997 von David Karger und Kollegen für Web-Caches eingeführt, platziert Knoten und Schlüssel gleichermaßen auf einem kreisförmigen Hash-Raum; jeder Schlüssel gehört zum nächsten Knoten im Uhrzeigersinn. Ein neuer Knoten übernimmt nur die Schlüssel zwischen sich und seinem Vorgänger, etwa 1/N des Ganzen, und ein entfernter Knoten übergibt seine Schlüssel an seinen Nachbarn. Bei wenigen Knoten sind

die Bögen ungleich, daher erhält jeder physische Knoten viele *virtuelle Knoten* (Tokens) an verschiedenen Stellen des Rings, was die Last ausgleicht und einen neuen Knoten von jedem bestehenden ein wenig übernehmen lässt. Dynamo und Cassandra verwenden dieses Schema.

Konsistentes Hashing auf einem Ring



Schlüssel gehören zum nächsten Knoten im Uhrzeigersinn, daher übernimmt ein neuer Knoten einen Bogen, statt alles neu zu mischen.

Zwei weitere Rebalancing-Strategien sind verbreitet. Bei einer **festen Anzahl von Partitionen** legen Sie zu Beginn viel mehr Partitionen an, als es Knoten gibt (etwa 1024 für 10 Knoten), und weisen den Knoten ganze Partitionen zu; ein neuer Knoten verschiebt ein paar ganze Partitionen, Schlüssel werden nie neu gehasht. Elasticsearch-Shards, Couchbase-vBuckets und viele selbst gebaute Sharding-Schemata nutzen das, und die wichtigste Entscheidung ist eine Partitionsanzahl, die groß genug für künftiges Wachstum ist. Bei **dynamischer Partitionierung**, wie in HBase und MongoDB, teilt sich eine Partition, wenn sie eine Größenschwelle überschreitet, und die Hälften können auf verschiedene Knoten wandern. Bei jedem Schema verschiebt Rebalancing große Datenmengen über das Netzwerk, während das System Verkehr bedient, daher ziehen es viele Teams vor, dass ein Mensch jede Verschiebung genehmigt, und drosseln sie, um die Latenz zu schützen.

Anfragen müssen außerdem zur richtigen Partition geleitet werden: Entweder nimmt jeder Knoten eine Anfrage an und leitet sie weiter (Cassandra-Knoten lernen den Ring per Gossip), oder eine Routing-Schicht kennt die Partitionszuordnung (`mongos` bei MongoDB), oder die Client-Bibliothek hält die Zuordnung. Die Zuordnung selbst liegt meist in einem Koordinationsdienst wie etcd oder ZooKeeper oder in der eigenen Konsensgruppe der Datenbank.

WENN DAS NEU FÜR SIE IST

Schritt für Schritt: einen Knoten hinzufügen, ohne alles zu verschieben

Der schwierigste Gedanke dieses Abschnitts ist, warum der naheliegende Weg, Schlüssel auf Knoten zu verteilen, zusammenbricht, wenn ein Knoten hinzukommt, und wie konsistentes Hashing das vermeidet. Verfolgen Sie die Bestellungen der Essenslieferplattform, während ein Cluster wächst.

- Eine Hashfunktion macht aus dem Schlüssel jeder Bestellung eine Zahl. Angenommen, die Zahlen laufen der Einfachheit halber von 0 bis 11, und eine Bestellung geht an Knoten „hash mod 3“, den Rest nach Division durch 3. Bestellung 7 liegt auf Knoten 1.
- Ein vierter Knoten kommt hinzu, und die Regel wird zu „hash mod 4“. Bestellung 7 gehört jetzt auf Knoten 3.
- Von den zwölf Zahlen ergeben nur 0, 1 und 2 für 3 und 4 denselben Rest. Die übrigen neun Bestellungen, 75 % davon, müssen auf einen anderen Knoten kopiert werden, nur um eine Maschine hinzuzufügen.
- Konsistentes Hashing platziert stattdessen Knoten und Schlüssel auf einem Kreis von Hashwerten, und jeder Schlüssel gehört zum nächsten Knoten im Uhrzeigersinn. Im oben gezeichneten Kreis liegt der neue Knoten D zwischen B und C.
- Nur Schlüssel auf dem Bogen zwischen B und D wechseln den Besitzer, von C zu D. Jeder andere Schlüssel bleibt, wo er ist, sodass etwa der Anteil eines Knotens an den Daten wandert, nicht der größte Teil.
- Bei wenigen Punkten sind die Bögen ungleich, und D entlastet nur C; daher erhält jeder Server viele Punkte rund um den Kreis, virtuelle Knoten genannt, und ein neuer Server nimmt jedem bestehenden eine kleine Scheibe ab.
- **Die Lehre:** Beurteilen Sie eine Platzierungsregel danach, was passiert, wenn sich der Cluster ändert, nicht nur danach, wie gleichmäßig sie die Schlüssel heute verteilt. Konsistentes Hashing oder viele im Voraus angelegte Partitionen, die als Ganzes verschoben werden, halten die Verschiebung klein, wenn ein Knoten hinzukommt.

Hot Keys und Schieflage

Hashing verteilt *Schlüssel* gleichmäßig, nicht die *Last*. Ist ein Schlüssel weit aktiver als die übrigen, ist seine Partition unabhängig vom Schema heiß: das Konto einer prominenten Person in einem sozialen Netzwerk, das Flaggschiff-Restaurant einer landesweiten Kette während einer Aktion oder das eigene Abwicklungskonto einer Bank, das jede Zahlung berührt. Verwaltete Speicher machen die Grenzen explizit; Amazon DynamoDB etwa dokumentiert eine Obergrenze pro Partition von 3000 Read Units und 1000 Write Units pro Sekunde. Gegenmaßnahmen sind das Cachen heißer Lesevorgänge, das Aufteilen eines Hot Key durch

Anhängen eines kleinen zufälligen Suffixes (Schreibvorgänge auf `restaurant:88#0` bis `restaurant:88#9`, um den Preis, alle zehn zu lesen und zusammenzuführen), das Verlagern eines sehr großen Mandanten auf eigene Partitionen und das Aggregieren von Zählern im Arbeitsspeicher, bevor sie geschrieben werden. Alle erhöhen die Komplexität, also messen Sie zuerst, welche Schlüssel tatsächlich heiß sind.

Mandantenfähiges SaaS: Silo, Pool und Bridge

Angenommen, die Essenslieferplattform verkauft ihre Bestellsoftware auch an Restaurantketten, wobei jede Kette ein **Mandant** ist. Die Kernfrage lautet, was Mandanten teilen. Die AWS SaaS Lens nennt drei Modelle. Im **Silo**-Modell hat jeder Mandant eigene Ressourcen, vielleicht eine eigene Datenbank, was die stärkste Isolation und die höchsten Kosten bringt, mit vielen Datenbanken, die bereitzustellen und zu migrieren sind. Im **Pool**-Modell teilen sich Mandanten alles, und jede Zeile trägt eine Mandanten-ID, was im Betrieb am günstigsten und einfachsten ist. Das **Bridge**-Modell mischt beide je Komponente: Ein Service kann aus regulatorischen Gründen oder wegen lauter Nachbarn als Silo laufen, ein anderer im Pool. Viele Produkte mischen auch nach Kunde: Kleine Ketten bleiben im Pool, und eine landesweite Kette erhält eine eigene Datenbank.

In einem Pool zeigt ein einziger vergessener Filter einer Kette die Bestellungen ihres Konkurrenten. **Row-Level Security** verlagert die Prüfung in die Datenbank. In PostgreSQL begrenzt eine Policy wie `USING (tenant_id = current_setting('app.tenant_id')::uuid)` jede Abfrage auf die Tabelle. Die Fallstricke sind wichtig. Superuser und `BYPASSRLS`-Rollen überspringen Policies, ebenso Tabelleneigentümer, sofern nicht `FORCE ROW LEVEL SECURITY` gesetzt ist; die Anwendung muss sich also mit einer gewöhnlichen Rolle verbinden. Unter dem Transaction Pooling von PgBouncer wird ein sitzungsweites `SET` nicht unterstützt, setzen Sie den Mandanten also mit `SET LOCAL` innerhalb jeder Transaktion. Eindeutigkeits- und Fremdschlüsselprüfungen umgehen Policies, was verraten kann, dass ein Wert existiert. Behandeln Sie die Policy als letzte Absicherung, mit Tests, die versuchen, Mandantengrenzen zu überschreiten, und denken Sie daran, dass auch Cache-Schlüssel, Queues und Suchindizes den Mandanten brauchen.

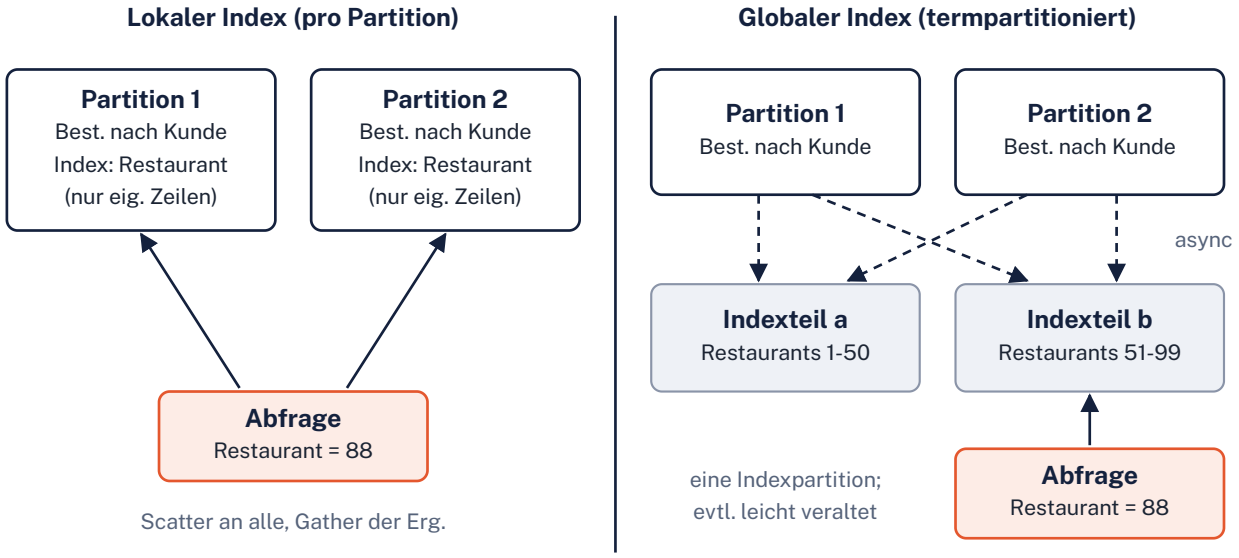
Ein **lauter Nachbar** ist ein Mandant, dessen Aktivität anderen schadet, etwa wenn die landesweite Werbeaktion einer Kette die gemeinsame Kapazität erschöpft. Microsofts Leitfaden behandelt das als Ressourcensteuerung: Kontingente und Rate Limits pro Mandant, Obergrenzen für teure Operationen, schwere Jobs außerhalb der Spitzenzeiten und Telemetrie mit Mandanten-ID, damit die Ursache sichtbar wird. Mandanten lassen sich dann neu verteilen oder auf dedizierte Kapazität verschieben.

Reicht eine Datenbank nicht aus, nutzt **mandantenbewusstes Sharding** die Mandanten-ID als Partitionsschlüssel. In Citus zum Beispiel trägt jede Tabelle die Mandantenspalte, die auch Teil von Primär- und Fremdschlüsseln ist, sodass die Zeilen eines Mandanten zusammenbleiben und Abfragen und Transaktionen eines einzelnen Mandanten als gewöhnliches PostgreSQL auf einem Knoten laufen. Hashing löst das Ungleichgewicht nicht, eine sehr große Kette braucht also womöglich einen eigenen Shard; der Rebalancer von Citus berücksichtigt nicht, welche Mandanten sich einen Shard teilen, und solche Verschiebungen brauchen Tooling. Berichte über alle Mandanten gehören in ein Data Warehouse. Entscheiden Sie auch die Wiederherstellung früh: Microsoft merkt an, dass die Wiederherstellung eines Mandanten aus einer gemeinsamen Datenbank bedeuten kann, die ganze Datenbank an anderer Stelle wiederherzustellen und die Daten dieses Mandanten gezielt zu übernehmen. Zellen führen dieselbe Idee für den gesamten Stack weiter.

Sekundärindizes über Partitionen hinweg

Für Abfragen über den Partitionsschlüssel funktioniert Partitionierung reibungslos. Abfragen über irgendetwas anderes brauchen einen Sekundärindex, und es gibt zwei Arten, ihn zu partitionieren. Ein **lokaler Index** (dokumentpartitioniert) liegt bei jeder Partition und deckt nur deren Zeilen ab. Schreibvorgänge ändern eine einzige Partition, aber eine Abfrage über das indexierte Attribut muss an jede Partition gesendet und die Ergebnisse müssen zusammengeführt werden (*Scatter-Gather*), sodass ihre Latenz von der langsamsten Partition bestimmt wird und ihre Kosten mit der Zahl der Partitionen wachsen. Ein **globaler Index** (termpartitioniert) ist selbst nach dem indexierten Wert partitioniert, sodass eine Abfrage eine Indexpartition liest; dafür muss ein einzelner Schreibvorgang nun womöglich Indexpartitionen auf anderen Knoten ändern, was meist asynchron geschieht, sodass der Index den Daten kurz hinterherhinken kann. DynamoDB zeigt beides: Ein lokaler Sekundärindex teilt den Partitionsschlüssel der Tabelle, ein globaler Sekundärindex hat einen eigenen und ist eventually consistent.

Lokale und globale Sekundärindizes



Ein lokaler Index macht Schreibvorgänge billig und Abfragen verstreut; ein globaler Index macht Abfragen gezielt und Indexänderungen entfernt und meist asynchron.

AUSGEARBEITETES BEISPIEL

Ausgearbeitetes Beispiel: einen Shard-Schlüssel für Bestellungen dimensionieren und wählen

Die Essenslieferplattform ist auf 30 Länder und 10 Millionen Bestellungen pro Tag gewachsen, mit einer abendlichen Spitze beim Dreifachen der durchschnittlichen Rate. Jede Bestellung erzeugt im Lauf ihres Lebens etwa 10 Schreibvorgänge (Anlage, Positionen, Statusänderungen, Zahlungsreferenzen) und belegt einschließlich Indizes etwa 4 KB. Kunden können ihre gesamte Bestellhistorie einsehen; Vorschriften verlangen, dass Bestellungen sechs Jahre lang abrufbar sind, der Support braucht aber nur für die letzten zwei Jahre Online-Zugriff.

Schreibrate	10.000.000 / 86.400 s = im Schnitt etwa 116 Bestellungen pro Sekunde; Spitze etwa 350 pro Sekunde; bei je 10 Schreibvorgängen etwa 3500 Schreibvorgänge pro Sekunde in der Spitze. Ein gut abgestimmter einzelner Primary kann das bewältigen; der Durchsatz allein erzwingt kein Sharding.
Speicherwachstum	10.000.000 x 4 KB = 40 GB pro Tag, etwa 14,6 TB pro Jahr, etwa 29 TB für zwei Jahre online.
Müssen wir sharden?	Müssten nur 90 Tage online sein (3,6 TB), wäre ein Primary mit Replikaten, mit zeitbasierter Tabellenpartitionierung und Archivierung ins Data Warehouse, die langweilige und richtige Antwort. Bei zwei Jahren online mit 29 TB werden die Zeiten für Wiederherstellung und Neuaufbau eines Replikats auf einem Knoten inakzeptabel, also shardet das Team.
Die Shards dimensionieren	Ziel sind etwa 2 TB pro Shard, damit sich ein Replikat in Stunden neu aufbauen lässt: 29 TB / 2 TB = etwa 15, also 16 Shards, jeder ein Leader mit zwei Followern. Legen Sie 1024 logische Partitionen an, die auf die 16 Shards abgebildet werden, sodass Wachstum das Verschieben ganzer Partitionen bedeutet, nicht neues Hashen. Spitzenschreibvorgänge pro Shard: 3500 / 16 = etwa 220 pro Sekunde.
Kandidat: order_id (Hash)	Perfekt gleichmäßige Schreibvorgänge, aber die Historie eines Kunden und die Live-Warteschlange eines Restaurants werden beide zu Scatter-Gather über 16 Shards. Als primärer Schlüssel für die Platzierung abgelehnt.
Kandidat: placed_at (Bereich)	Hervorragend für Berichte über Zeiträume, aber jede neue Bestellung schreibt in die neueste Partition: ein garantierter Hotspot. Abgelehnt.
Kandidat: restaurant_id	Die Warteschlange eines Restaurants ist lokal, aber große Ketten und Aktionen erzeugen Hot Keys, und die Kundenhistorie wird verstreut. Abgelehnt.
Kandidat: Land oder Stadt	Unterstützt Datenresidenz und lokale Abfragen, aber eine große Stadt kann 15 % aller Bestellungen tragen: Schieflage. Nur als grobe erste Ebene für die Datenresidenz beibehalten (Cluster in der EU und außerhalb der EU), nicht als Shard-Schlüssel.

Entscheidung	Sharding nach `hash(customer_id)`, mit Zeilen, die innerhalb jeder Partition nach `placed_at` sortiert sind. Die Bestell-ID enthält die Partitionsnummer des Kunden, sodass eine Suche nach Bestell-ID direkt zu einem Shard geht.
Die übrigen Abfragen abdecken	Die Live-Warteschlange des Restaurants wird von einem separaten Lesemodell bedient, das nach Restaurant geschlüsselt ist und aus Bestellereignissen aufgebaut wird (ein bewusster, eventually consistent globaler Index). Analysen laufen im Data Warehouse. Kundenübergreifende Transaktionen werden nicht gebraucht: Jede Bestellung betrifft einen Kunden.

Die Zahlen haben die Frage verändert. Der Durchsatz hat Sharding nie erfordert; die online vorgehaltenen Daten schon. Ein kleineres Online-Zeitfenster hätte den Bedarf ganz beseitigt, und das sollte man mit dem Fachbereich besprechen, bevor man ein geshardetes System baut.

Der Shard-Schlüssel wurde anhand der wichtigsten Abfragen und ihrer Konsistenzanforderungen gewählt, nicht allein anhand des Datenmodells. Jede andere Abfrage erhielt einen expliziten, eigens entworfenen Weg.

Die Wahl eines Shard-Schlüssels gehört zu den am schwersten umkehrbaren Entscheidungen: Ihn später zu ändern heißt, jede Zeile zu verschieben. Halten Sie ihn in einem ADR fest, mit den Mengen und Abfragen, von denen er ausging.

Häufige Fehler

Replikate als Backups behandeln. Ein Replikat kopiert ein irrtümliches `DELETE` getreulich innerhalb von Millisekunden; Backups und Point-in-Time-Recovery sind weiterhin nötig (Kapitel 0). **Standardmäßig von Replikaten lesen**, ohne zu entscheiden, welche Abfragen Verzögerung vertragen. **Einem automatischen Failover vertrauen, das nie geprobt wurde** oder das einen Knoten befördern kann, während der alte Leader noch Schreibvorgänge annimmt. **Zu früh sharden**: Eine geshardete Datenbank erschwert Joins, Transaktionen, Schemaänderungen und den Betrieb, und moderne Einzelknoten sind sehr groß; schöpfen Sie zuerst Indizierung, Caching, Read Replicas, Archivierung und vertikale Skalierung aus. **Nach einem monoton steigenden Schlüssel sharden**, was jedes Insert auf eine Partition konzentriert. **Sekundäre Zugriffswege vergessen** und erst nach der Aufteilung feststellen, dass eine wichtige Abfrage nun jeden Shard fragen muss. Wie der Rest dieses Kapitels zeigt, beruhen die guten Alternativen oft auf abgeleiteten Daten: Lesemodelle, Caches und analytische Speicher, jeweils für eine Art von Abfrage gebaut.

PRÜFEN SIE IHR VERSTÄNDNIS

▼ **Warum riskiert asynchrone Replikation den Verlust von Schreibvorgängen, die den Clients als festgeschrieben gemeldet wurden?**

Der Leader bestätigt, bevor irgendein Follower die Änderung hat; fällt er aus, bevor er sie versendet, hat der beförderte Follower diese Schreibvorgänge nie erhalten.

▼ **Ein Nutzer speichert eine Änderung und sieht sofort den alten Wert. Welche Garantie ist verletzt, und was ist eine Lösung?**

Read-your-Writes. Bedienen Sie die Lesevorgänge dieses Nutzers auf kürzlich geänderte Daten vom Leader, oder warten Sie, bis der Follower die Log-Position des Schreibvorgangs angewendet hat.

▼ **Warum ist hash mod N ein schlechter Weg, Schlüssel Knoten zuzuordnen?**

Eine Änderung von N verschiebt die meisten Schlüssel (etwa 80 % beim Schritt von 4 auf 5 Knoten), während konsistentes Hashing oder feste logische Partitionen nur etwa 1/N verschieben.

▼ **Was ist die Abwägung zwischen lokalen und globalen Sekundärindizes?**

Lokale Indizes halten Schreibvorgänge auf einer Partition, zwingen Abfragen aber, jede Partition zu fragen; globale Indizes lassen Abfragen gezielt eine Indexpartition ansteuern, erfordern aber partitionsübergreifende, meist asynchrone Indexänderungen.

▼ **Probieren Sie es aus: Eine Plattform verarbeitet 6 Millionen Bestellungen pro Tag, von denen jede 10 Schreibvorgänge auslöst, und die Spitze liegt beim Dreifachen des Durchschnitts. Jeder Shard verkraftet sicher 500 Schreibvorgänge pro Sekunde. Wie viele Shards werden gebraucht? Vergleichen Sie Kunden-ID und Bestellzeitstempel als Shard-Schlüssel.**

Durchschnitt: 60.000.000 / 86.400 = 694 Schreibvorgänge pro Sekunde. Spitze: etwa 2083. Shards: 2083 / 500 = 4,2, also 5, plus Reserve für Wachstum. Ein Hash der Kunden-ID verteilt die Last gleichmäßig und hält die Historie eines Kunden zusammen, die wichtigste Abfrage. Ein Zeitstempel als Schlüssel schickt alle neuen Schreibvorgänge an einen Shard und erzeugt einen Hotspot. Berichte über Zeiträume brauchen dann Scatter-Gather oder, besser, einen getrennten Analysespeicher. Ergänzen Sie jeden Shard für die Verfügbarkeit um Replikate.

QUELLEN

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

CQRS und Event-Sourcing

BEVOR SIE DAS LESEN

Dieser Abschnitt baut auf:

- 5.7 Domain-Driven Design
- 6.4 Ereignisgesteuert
- 7.3 Transaktionen

Begriffe von früher, die er verwendet: Aggregat, Bounded Context, Eventual Consistency, Optimistische Nebenläufigkeitskontrolle

DIE IDEE IN KÜRZE

Eine Bank führt ein Hauptbuch, in das jede Einzahlung und jede Abhebung eingetragen wird, eine Zeile nach der anderen, und nichts wird je ausradiert; Fehler werden durch eine Korrekturzeile behoben. Ihr Kontostand ist einfach die Summe aller Zeilen. Ihr monatlicher Kontoauszug ist eine übersichtliche Zusammenfassung, die aus diesem Hauptbuch zum bequemen Lesen erstellt wird. Event-Sourcing speichert Daten so wie das Hauptbuch: als Liste von Dingen, die geschehen sind. CQRS hält das Hauptbuch (für Änderungen) getrennt von den Auszügen (zum Nachschlagen). Beide sind mächtig, und beide bringen echte Komplexität mit sich.

WAS SIE WISSEN MÜSSEN

- Command-Query Separation (Bertrand Meyer, 1988) besagt, dass eine Methode entweder den Zustand ändern oder eine Antwort liefern sollte, nicht beides; CQRS (Greg Young, um 2010) wendet dieselbe Trennung auf ganze Modelle an.
- Bei CQRS setzt ein Schreibmodell Geschäftsregeln für Commands durch, während Projektionen ein oder mehrere denormalisierte Lesemodelle für Abfragen aktuell halten; Lesemodelle sind abgeleitete Daten, die sich neu aufbauen lassen.
- Event-Sourcing speichert jede Änderung als unveränderliches Ereignis in der Vergangenheitsform in einem nur anfügbaren Stream pro Aggregat; der aktuelle Zustand ist ein deterministischer Left Fold der Ereignisse.
- Event-Sourcing bietet einen vollständigen Audit-Trail, zeitbezogene Abfragen, neu aufbaubare und rückwirkende Lesemodelle sowie Replay zur Fehlersuche; Snapshots verkürzen lange Folds, sind aber nur ein Cache.

- Die Kosten sind real: Komplexität, Eventual Consistency der Lesemodelle, Versionierung von Ereignissen durch Upcasting oder neue Ereignistypen und das Löschen personenbezogener Daten aus einem unveränderlichen Log, meist durch Crypto-Shredding.
- Martin Fowler warnt, dass CQRS bei den meisten Systemen riskante Komplexität hinzufügt; wenden Sie beide Muster auf bestimmte Bounded Contexts mit wirklich unterschiedlichen Lese- und Schreibanforderungen an, nie standardmäßig auf ein ganzes System.

Von der Command-Query Separation zu CQRS

Die Idee fängt klein an. In *Object-Oriented Software Construction* (erste Auflage 1988) schlug Bertrand Meyer die Command-Query Separation vor: Jede Methode sollte entweder ein Command sein, das den Zustand ändert und nichts zurückgibt, oder eine Query, die eine Antwort liefert und nichts ändert. Eine Frage zu stellen, sollte nie die Antwort verändern. Martin Fowler weist darauf hin, dass die Regel sinnvolle Ausnahmen hat (ein Element von einem Stack zu nehmen, ändert den Stack und gibt zugleich das Element zurück), aber als Standard macht sie Code leichter nachvollziehbar, weil Queries frei aufgerufen werden können, in beliebiger Reihenfolge und ohne Seiteneffekte.

Um 2010 hob Greg Young das Prinzip von Methoden auf ganze Modelle und nannte das Ergebnis Command Query Responsibility Segregation (CQRS). Statt eines einzigen Objektmodells, und meist eines einzigen Satzes von Tabellen, für Änderungen und Lesezugriffe zugleich hat das System ein **Schreibmodell**, das Commands verarbeitet und Geschäftsregeln durchsetzt, und ein oder mehrere **Lesemodelle**, die auf die Fragen der Nutzer zugeschnitten sind. Die Einsicht dahinter: Schreiben und Lesen sind verschiedene Aufgaben. Schreiben braucht strenge Regeln, kleine Transaktionen und eine normalisierte Struktur, damit jede Tatsache nur einmal gespeichert ist. Lesen braucht Geschwindigkeit, denormalisierte Formen, die zu Bildschirmen und Berichten passen, und oft weit mehr Traffic. Ein einzelnes Modell, das beides gut machen will, macht meist beides schlecht, sobald die Domäne reichhaltig wird.

Was CQRS tatsächlich ändert

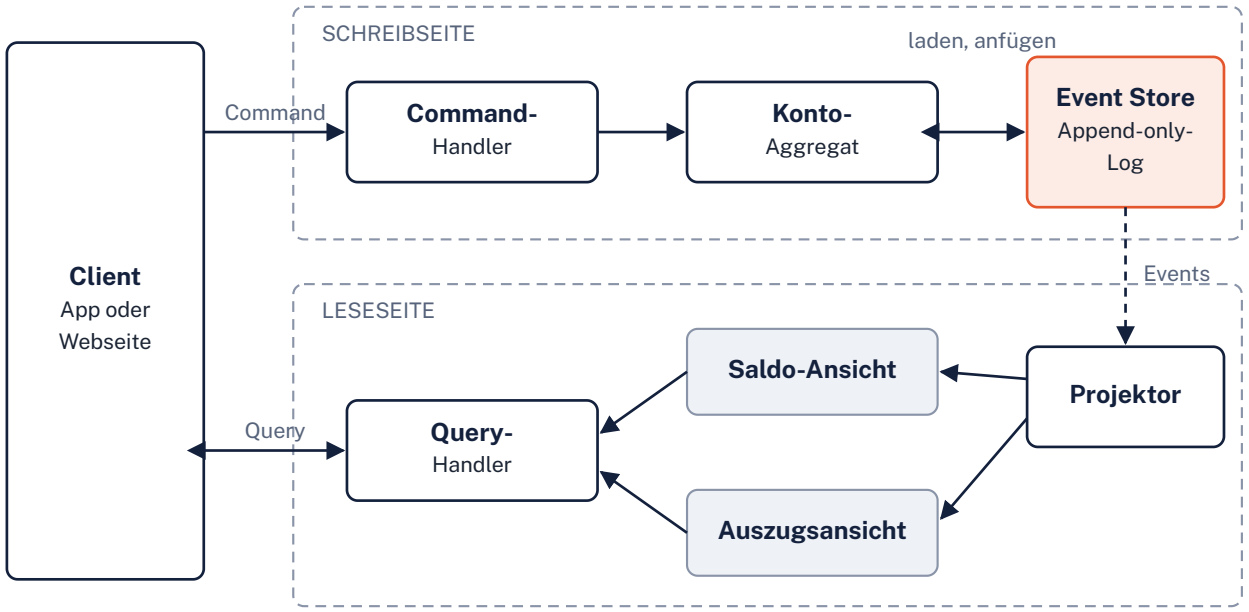
Auf der Schreibseite kommt ein Command wie `WithdrawMoney(account 42, £50)` bei einem Command-Handler an. Der Handler lädt das betreffende Aggregat (die Gruppe von Daten, die innerhalb einer Transaktion konsistent bleiben muss, ein Begriff aus dem Domain-Driven Design, behandelt in Abschnitt 5.7), lässt es die Geschäftsregeln anwenden und speichert das Ergebnis. Commands sind Anfragen und können abgelehnt werden: Das Konto kann gesperrt oder die Deckung unzureichend sein. Auf der Leseseite wird eine Query wie „Zeige mir die

letzten 30 Buchungen mit laufendem Kontostand“ aus einem Lesemodell beantwortet, das genau für diesen Bildschirm gebaut ist, ohne Joins, ohne Geschäftslogik und ohne Sperren auf den Schreibtabellen.

Irgendetwas muss die Lesemodelle aktuell halten. Diese Aufgabe übernimmt eine Projektion: Code, der auf Änderungen auf der Schreibseite lauscht und ein Lesemodell aktualisiert. Projektionen können eine Tabelle mit Kontoständen, einen Suchindex der Buchungen, ein Dokument mit dem Monatsauszug und einen Feature Store für die Betrugsbewertung pflegen, jeweils aus demselben Strom von Änderungen. Lesemodelle sind daher **abgeleitete Daten**: Sie lassen sich jederzeit verwerfen und aus der Schreibseite neu aufbauen. Genau diese Eigenschaft macht es sicher, nächstes Jahr ein neues Lesemodell hinzuzufügen oder die Form eines alten zu ändern.

CQRS gibt es in Abstufungen, und die leichteste Stufe ist oft alles, was ein System braucht. In der leichtesten Form sind Schreib- und Lesemodell verschiedene Klassen über derselben Datenbank, und Lesezugriffe werden über Views oder eigenen Abfragecode bedient. In der mittleren Form liegen die Lesemodelle in eigenen Tabellen, die in derselben Transaktion wie der Schreibvorgang aktualisiert werden. In der schwersten Form liegen die Lesemodelle in anderen Speichern (einer Suchmaschine, einer Dokumentdatenbank, einem Cache) und werden asynchron über Ereignisse aktualisiert, was Skalierung und Flexibilität bringt, aber auch Eventual Consistency. CQRS setzt weder Event-Sourcing noch einen Message Broker noch Microservices voraus; das sind getrennte Entscheidungen, die oft, aber nicht zwingend, zusammen getroffen werden.

CQRS mit Event-Sourcing



Commands und Queries nehmen verschiedene Wege; der Event Store ist die Quelle der Wahrheit, und jedes Lesemodell wird von einem Projektor daraus abgeleitet.

WENN DAS NEU FÜR SIE IST

Wenn das neu für Sie ist

Denken Sie an das Hauptbuch einer Bank und an einen Kontoauszug. Das Hauptbuch ist der Ort, an dem jede Einzahlung und Abhebung nach strengen Regeln eingetragen wird, Zeile für Zeile. Der Auszug ist eine Zusammenfassung, die aus dem Hauptbuch in einer gut lesbaren Form erstellt wird. Sie schreiben nie auf den Auszug, um Geld zu bewegen, und Sie lesen nie das Hauptbuch Zeile für Zeile, um Ihren Kontostand zu finden. CQRS gibt Software dieselbe Trennung: ein Modell zum Ändern, andere Modelle zum Nachschlagen. Event-Sourcing geht einen Schritt weiter und macht das Hauptbuch selbst zum Einzigen, was dauerhaft gespeichert wird.

Event-Sourcing: Zustand als Folge von Tatsachen

Die meisten Systeme speichern den aktuellen Zustand. Hebt Asha £120 ab, ändert sich die Zeile in `accounts` von £500 auf £380, und der alte Wert ist weg, sofern nicht jemand zusätzlich eine Audit-Zeile schreibt. Event-Sourcing, 2005 von Martin Fowler beschrieben, kehrt das um: Das System speichert jede Änderung als unveränderliches Ereignis in einem nur

anfügbaren Log, und der aktuelle Zustand ist das, was man erhält, wenn man die Ereignisse der Reihe nach erneut abspielt. Die Ereignisse sind die Quelle der Wahrheit; der Kontostand ist eine Berechnung.

Ereignisse werden in der Vergangenheitsform benannt, weil sie Tatsachen festhalten, die bereits geschehen und akzeptiert sind: `AccountOpened`, `MoneyDeposited`, `MoneyWithdrawn`. Ein Command (`WithdrawMoney`) ist eine Anfrage, die abgelehnt werden kann; ein Ereignis (`MoneyWithdrawn`) lässt sich nicht mehr ablehnen, nur durch ein späteres Ereignis ausgleichen. Jedes Aggregat hat seinen eigenen Event-Stream, nummeriert ab Version 1, und der Store garantiert, dass die Ereignisse eines Streams in ihrer Reihenfolge bleiben.

Den Zustand neu aufzubauen, ist ein **Left Fold**: Man beginnt mit nichts, wendet das erste Ereignis an, wendet das zweite auf das Ergebnis an und so weiter. Die Apply-Funktion muss rein und deterministisch sein: Sie darf weder eine Uhr noch einen Zufallszahlengenerator noch einen anderen Service aufrufen, weil das erneute Abspielen derselben Ereignisse immer denselben Zustand ergeben muss. Jeder Wert, der von außen kam (ein Wechselkurs, ein Zeitstempel, eine Betrugsbewertung), wird im Ereignis festgehalten, wenn er zum ersten Mal festgelegt wird.

Das Schreiben folgt einem festen Ablauf. Der Command-Handler lädt den Stream, faltet ihn zum aktuellen Zustand, führt die Geschäftsregel aus und fügt die neuen Ereignisse mit der Version an, die er zu schreiben erwartet. Hat in der Zwischenzeit ein anderes Command an denselben Stream angefügt, schlägt das Anfügen fehl, und der Handler versucht es mit frischem Zustand erneut. Das ist die optimistische Nebenläufigkeitskontrolle aus Abschnitt 7.3, angewendet pro Stream, und deshalb können zwei gleichzeitige Abhebungen nicht beide dieselben £100 ausgeben.

AUSGEARBEITETES BEISPIEL

Ausgearbeitetes Beispiel: ein Bankkonto als Ereignisse

Ashas Girokonto ist der Stream `account-42`. Beträge werden in Pence gespeichert, wie Abschnitt 0.2 es für Geld empfiehlt. Fünf Ereignisse wurden angefügt; dann versucht sie, £200 abzuheben.

v1 AccountOpened (Dispositionslimit £0)	Zustand: Kontostand £0,00, Version 1
v2 MoneyDeposited £500,00	£0,00 + £500,00 = £500,00, Version 2
v3 MoneyWithdrawn £120,00	£500,00 − £120,00 = £380,00, Version 3
v4 MoneyDeposited £75,50	£380,00 + £75,50 = £455,50, Version 4
v5 MoneyWithdrawn £300,00	£455,50 − £300,00 = £155,50, Version 5
Command: WithdrawMoney £200,00	Regel: Kontostand minus Betrag darf nicht unter minus das Limit fallen. £155,50 − £200,00 = −£44,50, unter £0. Command abgelehnt; kein Ereignis wird angefügt.
Command: WithdrawMoney £20,00	£155,50 − £20,00 = £135,50 ist zulässig. Der Handler fügt v6 MoneyWithdrawn £20,00 an und erwartet, dass Version 6 frei ist.
Zeitbezogene Abfrage: Kontostand nach v3	Nur v1 bis v3 falten: £380,00. Keine Historientabelle nötig.
Audit-Frage: Warum beträgt der Kontostand £135,50?	Die sechs Ereignisse sind die vollständige Antwort, in Reihenfolge, und in jedem ist festgehalten, wer, wann und warum.

Das abgelehnte Command hat im Stream keine Spur hinterlassen. Muss die Bank Ablehnungen festhalten (etwa für die Betrugsanalyse), fügt sie ein eigenes Ereignis `WithdrawalRefused` an, das den Kontostand nicht ändert.

Die doppelte Buchführung ist das älteste ereignisbasierte System überhaupt: Buchhalter radieren nie eine Zeile aus, sie buchen eine Korrekturbuchung. Bankdomänen passen daher ganz natürlich zu Event-Sourcing, weshalb es oft an einem Konto erklärt wird.

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

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

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

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

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

Den Stream zum aktuellen Zustand falten, ein Command anhand dieses Zustands entscheiden und eine zeitbezogene Abfrage beantworten, indem nur ein Teil des Streams gefaltet wird.

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

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

Ein minimaler Event Store in einer relationalen Datenbank: Der zusammengesetzte Primärschlüssel macht ein gleichzeitiges Anfügen zu einem erkennbaren Konflikt.

Falten, Snapshots und zeitbezogene Abfragen



Zeitbezogene Abfrage: Der Stand zu v3 ist der Fold von nur v1 bis v3, also £380,00.

Der aktuelle Zustand ist der Fold aller Ereignisse; ein Snapshot verkürzt den Fold, und das Falten eines Anfangsstücks des Streams beantwortet Fragen über die Vergangenheit.

Was Event-Sourcing Ihnen bringt

Ein vollständiger Audit-Trail schon durch die Bauweise. In einem zustandsbasierten System ist das Audit-Log ein zweiter Schreibvorgang, den jemand vergessen kann und der den Daten widersprechen kann. In einem System mit Event-Sourcing ist das Log die Daten, also kann es nicht auseinanderlaufen. Aufsichtsbehörden im Bank-, Versicherungs- und Gesundheitswesen schätzen das sehr.

Zeitbezogene Abfragen. „Wie hoch war der Kontostand am 31. März um 23:59 Uhr?“ oder „Welche Bestellungen waren im Status PREPARING, als die Küchenanzeige ausfiel?“ werden zu Folds über ein Anfangsstück jedes Streams statt zu forensischer Arbeit an Backups.

Neuaufbau und neue Lesemodelle. Weil Lesemodelle abgeleitet sind, kann ein Team einen Fehler in einer Projektion beheben, deren Ausgabe löschen und alle Ereignisse erneut abspielen, um eine korrigierte Ansicht zu erzeugen. Es kann auch ein Lesemodell bauen, an das beim ersten Aufzeichnen der Ereignisse niemand gedacht hat, etwa einen Bericht darüber, wie oft Kunden innerhalb einer Stunde nach dem Gehaltseingang Bargeld abheben, über die gesamte Historie.

Fehlersuche und Was-wäre-wenn-Analysen. Einen Stream in eine Testumgebung zu kopieren und ihn erneut abzuspielen, reproduziert genau den Zustand, der einen Fehler verursacht hat. Dieselben Ereignisse mit geänderten Regeln abzuspielen, zeigt, was die neuen Regeln getan hätten.

Eine natürliche Quelle für Integrationsereignisse. Die Ereignisse können andere Services speisen, wobei die in einem Stream gespeicherten Ereignisse am besten als internes Detail des besitzenden Service behandelt werden. Sie direkt zu veröffentlichen, koppelt jeden Consumer an das interne Modell; viele Teams übersetzen sie stattdessen in eine kleinere Menge öffentlicher Integrationsereignisse, gestaltet als die langlebigen veröffentlichten Verträge, die Abschnitt 6.4 beschreibt.

Snapshots und lange Streams

Fünf Ereignisse zu falten, geht sofort; 400.000 Ereignisse eines umsatzstarken Händlerkontos bei jedem Command zu falten, nicht. Ein Snapshot speichert den gefalteten Zustand eines Streams bei einer bekannten Version, sodass das Laden zu „den neuesten Snapshot lesen und dann nur die Ereignisse danach anwenden“ wird. Snapshots sind ein Cache, keine Quelle der Wahrheit: Sie lassen sich jederzeit löschen und im Hintergrund neu erzeugen, ohne das System anzuhalten. Erstellen Sie sie alle N Ereignisse oder nach Zeitplan, und speichern Sie die Version des Apply-Codes mit, denn ein Snapshot aus altem Code ist falsch, sobald sich die Fold-Logik ändert.

Bevor Sie zu Snapshots greifen, prüfen Sie das Stream-Design. Sehr lange Streams bedeuten oft, dass das Aggregat zu groß ist. Ein Bankkonto, das 40 Jahre besteht, lässt sich als Folge kürzerer Streams modellieren, einer pro Auszugszeitraum, jeder eröffnet mit einem Ereignis `BalanceCarriedForward`, das den Schlusssaldo des vorigen Zeitraums festhält. Das entspricht dem Jahresabschluss in der Buchhaltung und hält jeden Stream kurz genug, um schnell geladen zu werden.

Die Kosten, ehrlich benannt

Komplexität und Ungewohntheit. Event-Sourcing verändert, wie alle Entwickler im Team über Persistenz denken. Einfache Fragen wie „Welche Kunden wohnen in Leeds?“ lassen sich dem Event Store gar nicht stellen; sie brauchen eine Projektion. Werkzeuge zum Untersuchen, Migrieren und Reparieren von Event Stores sind weniger ausgereift als für relationale Tabellen, und weniger Fachleute haben ein solches System schon in Produktion betrieben.

Eventual Consistency der Lesemodelle. Laufen Projektionen asynchron, kann ein Nutzer eine Abhebung vornehmen und danach einen Kontostand sehen, der sie noch nicht enthält. Die üblichen Abhilfen: die neue Stream-Version vom Command zurückgeben und die Leseseite warten lassen, bis ihre Projektion diese Version verarbeitet hat (eine Read-your-Writes-Garantie wie in Abschnitt 7.4); die Benutzeroberfläche optimistisch aus dem Ergebnis des Commands aktualisieren; oder kritische Lesemodelle synchron in derselben Transaktion wie das Anfügen aktualisieren. Projektionen müssen außerdem idempotent sein und die verarbeitete Position festhalten, weil Ereignisse nach einem Absturz mehr als einmal zugestellt werden können.

Versionierung von Ereignissen. Ereignisse werden für immer gespeichert, also lässt sich ihr Schema nie einfach ändern. Bekommt `MoneyDeposited` ein Währungsfeld, fehlt es in Jahren alter Ereignisse weiterhin. Zu den Standardtechniken, die Greg Young in *Versioning in an Event Sourced System* zusammengestellt hat, gehören Tolerant Reader, die für fehlende Felder Standardwerte liefern, Upcasting (eine Funktion, die eine alte Ereignisversion beim Lesen in die aktuelle Form umwandelt, sodass der Fachcode nur eine Version sieht), die Einführung eines neuen Ereignistyps neben dem alten und, als letztes Mittel, das Kopieren eines ganzen Streams in einen neuen mit umgewandelten Ereignissen. Abschnitt 7.7 behandelt die allgemeinen Kompatibilitätsregeln.

Personenbezogene Daten löschen. Ein unveränderliches Log kollidiert mit dem Recht auf Löschung nach dem Datenschutzrecht. Die gängige Technik ist Crypto-Shredding: Personenbezogene Felder in Ereignissen werden mit einem Schlüssel pro Kunde verschlüsselt, der in einem separaten Schlüsselspeicher liegt, und Löschen bedeutet, diesen Schlüssel zu vernichten, sodass der Geheimtext nicht mehr lesbar ist. Die Alternative ist, personenbezogene Daten ganz aus den Ereignissen herauszuhalten und nur einen Verweis auf einen löschbaren Datensatz zu speichern. Ob die Vernichtung eines Schlüssels als Löschung gilt, ist eine rechtliche und keine technische Beurteilung; stimmen Sie den Ansatz daher mit dem Datenschutzbeauftragten (DSB) ab; Abschnitt 8.7 behandelt die weiteren Pflichten.

Betriebsaufwand. Eine Projektion über eine Milliarde Ereignisse neu aufzubauen, kann Stunden dauern; Teams brauchen daher einen Weg, die neue Projektion neben der alten laufen zu lassen und umzuschalten, wenn sie aufgeholt hat. Event Stores wachsen unbegrenzt, und mit ihnen wachsen Speicher-, Backup- und Replay-Zeiten.

ACHTUNG

Fowlers Warnung

Martin Fowlers Artikel über CQRS von 2011 ist ungewöhnlich deutlich: Er schreibt, dass „CQRS bei den meisten Systemen riskante Komplexität hinzufügt“, und empfiehlt, es nur für bestimmte Teile eines Systems einzusetzen, typischerweise einen Bounded Context mit wirklich unterschiedlichen Lese- und Schreibanforderungen, nie als Architektur des ganzen Systems. Für Event-Sourcing gilt das umso mehr. Teams, die beides überall eingesetzt haben, weil es Mode war, berichten von Jahren unnötiger Komplexität, vor allem in einfachen CRUD-Bereichen (Anlegen, Lesen, Ändern, Löschen) wie Stammdaten oder Nutzerprofilen.

ENTSCHEIDUNGSHILFE

Entscheidungshilfe: Sollte dieser Teil des Systems CQRS oder Event-Sourcing nutzen?

Stellen Sie diese Fragen für einen Bounded Context, nicht für das ganze System.

Q1 **Besteht dieser Bereich überwiegend aus einfachem Anlegen, Lesen, Ändern und Löschen mit wenigen Geschäftsregeln?**

- **Ja** Nutzen Sie ein konventionelles Modell mit einer Datenbank. Keines der beiden Muster rechnet sich hier.
- **Nein** Weiter: Die Domäne hat echtes Verhalten, das sich zu modellieren lohnt.

Q2 **Unterscheiden sich Lese- und Schreibzugriffe stark in Form oder Menge (zum Beispiel viele denormalisierte Bildschirme, Suche oder tausendmal mehr Lese- als Schreibzugriffe)?**

- **Ja** Ziehen Sie CQRS in Betracht. Beginnen Sie mit der leichtesten Form: eigener Lesecode oder Views über derselben Datenbank.
- **Nein** Behalten Sie ein einziges Modell; ergänzen Sie eine Read Replica oder einen Cache, wenn allein die Leselast das Problem ist.

Q3 **Ist die Historie selbst wertvoll: regulatorisches Audit, zeitbezogene Abfragen, erneutes Abspielen mit korrigierter Logik oder viele künftige Lesemodelle aus denselben Tatsachen?**

- **Ja** Ziehen Sie Event-Sourcing für diesen Kontext in Betracht, idealerweise in einer hauptbuchartigen Domäne wie Konten, Zahlungen oder Bestelllebenszyklen.
- **Nein** Speichern Sie den aktuellen Zustand; veröffentlichen Sie Ereignisse über eine Outbox, wenn andere Services von Änderungen erfahren müssen.

Q4 **Hat das Team die Erfahrung, die Werkzeuge und die Zeit, um Ereignisversionierung, den Neuaufbau von Projektionen und Eventual Consistency zu bewältigen?**

- **Ja** Machen Sie weiter, halten Sie den Bereich mit Event-Sourcing klein, und halten Sie die Entscheidung und ihre Gründe in einem ADR fest.
- **Nein** Speichern Sie den aktuellen Zustand plus eine nur anfügbare Audit-Tabelle; das bringt den größten Teil des Audit-Nutzens zu einem Bruchteil der Kosten.

Wann keines von beiden passt, und häufige Fehler

Keines der beiden Muster passt zu einer Domäne, die überwiegend aus Formularen über Daten besteht, zu einem kleinen Team, das seine erste Version baut, oder zu einem System, dessen einziges Problem die Leselast ist (eine Read Replica oder ein Cache ist weit günstiger). Event-

Sourcing passt auch schlecht, wo Daten auf Verlangen in großer Zahl wirklich gelöscht werden müssen oder wo die meisten Abfragen viele Aggregate übergreifen, weil jede solche Abfrage eine eigene Projektion braucht.

Die häufigen Fehler sind vorhersehbar. Teams wenden die Muster systemweit an statt auf einen Kontext. Sie machen gespeicherte Ereignisse zum öffentlichen Vertrag und frieren damit ihr internes Modell ein. Sie bringen abgeleitete Werte oder externe Aufrufe in die Apply-Funktion, sodass das erneute Abspielen andere Antworten liefert. Sie gestalten Ereignisse als generische `AccountUpdated`-Blobs, die festhalten, dass sich etwas geändert hat, aber nicht, was geschah oder warum, und verlieren damit den größten Teil des Nutzens. Und sie vergessen, dass ein Lesemodell ein Cache ist: Sie fangen an, direkt hineinzuschreiben, und danach lässt es sich nicht mehr neu aufbauen.

Gezielt eingesetzt, ist die Kombination mächtig. Eine Essenslieferplattform könnte den Bestelllebenszyklus mit Event-Sourcing abbilden (aufgegeben, angenommen, in Zubereitung, abgeholt, geliefert, erstattet), weil Streitfälle und Erstattungen die genaue Historie brauchen, und Speisekarten, Restaurantprofile und Nutzereinstellungen als gewöhnliche Tabellen führen. Eine Bank könnte das Hauptbuch und die Zahlungen mit Event-Sourcing abbilden und die Kontaktdaten der Kunden konventionell speichern, was zugleich die meisten personenbezogenen Daten aus dem unveränderlichen Log heraushält.

PRÜFEN SIE IHR VERSTÄNDNIS

▼ Warum darf die Funktion, die Ereignisse auf den Zustand anwendet, nie die Systemuhr aufrufen?

Weil das erneute Abspielen derselben Ereignisse immer denselben Zustand ergeben muss. Eine Uhr (oder eine Zufallszahl oder ein externer Aufruf) würde beim Replay andere Antworten liefern; jeder solche Wert muss im Ereignis festgehalten werden, wenn er zum ersten Mal festgelegt wird.

▼ Was ist der Unterschied zwischen dem Command WithdrawMoney und dem Ereignis MoneyWithdrawn?

Das Command ist eine Anfrage, die von den Geschäftsregeln abgelehnt werden kann; das Ereignis ist eine festgehaltene Tatsache, die bereits akzeptiert wurde und sich nur durch ein späteres Ereignis ausgleichen, nie entfernen lässt.

▼ Eine Projektion hat einen Fehler, der drei Monate lang falsche Kontoauszüge erzeugt hat. Wie hilft Event-Sourcing?

Weil das Lesemodell abgeleitet ist, behebt das Team die Projektion, baut durch erneutes Abspielen aller Ereignisse ein neues Lesemodell auf (meist neben dem alten) und schaltet um, ohne Daten von Hand reparieren zu müssen.

▼ Ist ein Snapshot Teil der Quelle der Wahrheit?

Nein. Er ist ein zwischengespeicherter Fold eines Streams bei einer bestimmten Version, der sich jederzeit löschen und aus den Ereignissen neu erzeugen lässt.

▼ Probieren Sie es aus: Die Architektin möchte Event-Sourcing überall einsetzen. Betrachten Sie ein Treuepunktekonto, das einen vollständigen Audit-Trail braucht, und die Tabelle mit den Öffnungszeiten der Veranstaltungsorte. Würden Sie es jeweils anwenden? Nennen Sie zwei Kosten, auf die Sie beim Treuepunktekonto hinweisen würden.

Verwenden Sie es, wenn überhaupt, für das Hauptbuch: Eine Append-only-Historie liefert den Audit-Trail, und Salden lassen sich durch Abspielen der Ereignisse neu aufbauen. Öffnungszeiten der Veranstaltungsorte sind einfaches CRUD, eine schlichte Tabelle genügt, und Event-Sourcing würde nur Risiko hinzufügen. Weisen Sie beim Hauptbuch auf die Versionierung der Ereignisse hin, da alte Ereignisse für immer bleiben, und auf das Löschen personenbezogener Daten aus einem unveränderlichen Log, meist per Crypto-Shredding. Auch Lesemodelle sind nur letztlich konsistent. Wenden Sie das Muster pro Kontext an, nie standardmäßig.

QUELLEN

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

Caching-Strategien und ihre Fallstricke

BEVOR SIE DAS LESEN

Dieser Abschnitt baut auf:

- 0.2 Daten und Algorithmen
- 0.4 Wie Computer kommunizieren
- 2.4 Performance und Skalierbarkeit

Begriffe von früher, die er verwendet: Latenz, Durchsatz, HTTP, System of Record, Read-your-Writes

DIE IDEE IN KÜRZE

Stellen Sie sich ein belebtes Café vor. Statt für jede Bestellung Bohnen zu mahlen, hält der Barista ein kleines Glas frisch gemahlenen Kaffee auf der Theke bereit. Die meisten Bestellungen werden in Sekunden aus dem Glas bedient; erst wenn es leer ist, geht jemand ins Lager. Das Glas ist ein Cache. Es macht den Service schnell und schont das Lager, wirft aber Fragen auf: Wie lange kann gemahlener Kaffee stehen, bevor er schal schmeckt, was passiert, wenn das Glas im morgendlichen Andrang leer wird, und wer denkt daran, es auszuleeren, wenn die Bohnen wechseln? Software-Caches werfen genau dieselben Fragen auf.

WAS SIE WISSEN MÜSSEN

- Ein Cache tauscht eine zweite, möglicherweise veraltete Kopie der Daten gegen geringere Latenz, höheren Durchsatz und Schutz der Quelle; legen Sie zuerst fest, was veraltet sein darf, wie stark und für wen.
- Cache-Aside ist der sichere Standard; Read-Through verlagert das Laden in die Cache-Schicht; Write-Through hält den Cache aktuell, auf Kosten langsamerer Schreibvorgänge; Write-Behind riskiert Datenverlust und darf nie Geldbeträge halten; Refresh-Ahead passt zu wenigen heißen Schlüsseln.
- Kombinieren Sie Invalidierung bei Änderungen mit einer TTL als Sicherheitsnetz; aktualisieren Sie bei Schreibvorgängen die Datenbank und löschen Sie dann den Cache-Eintrag, statt ihn zu überschreiben.
- Verhindern Sie Stampedes mit Request Coalescing, gestreutem Ablauf (Jitter), Stale-While-Revalidate, probabilistischer vorzeitiger Aktualisierung und Cache Warming.

- Die Trefferquote bestimmt die Datenbanklast weit stärker als die durchschnittliche Latenz: Jede zusätzliche Neun an Treffern teilt die Datenbankabfragen etwa durch zehn, und ein kalter Cache kann eine Datenbank überfordern, die für gecachten Traffic dimensioniert ist.
- Caches gibt es in Browsern, CDNs, Reverse Proxys, Anwendungen und Datenbanken; HTTP steuert sie mit Cache-Control, ETag und bedingten Anfragen (RFC 9111, RFC 9110).
- LRU und LFU sind die üblichen Verdrängungsstrategien; Schlüssel müssen alles enthalten, was die Antwort verändert, und personenbezogene oder finanzielle Daten gehören nicht in gemeinsam genutzte Caches.

Warum cachen, und was es kostet

Ein Cache hält eine Kopie von Daten an einem schnelleren oder näheren Ort als ihrem ursprünglichen Zuhause, damit wiederholte Anfragen nicht jedes Mal den vollen Preis zahlen. Abschnitt 0.2 hat die Idee als Tausch von Speicherplatz gegen Zeit eingeführt. Im Architekturmaßstab ist der Tausch interessanter, weil ein Cache drei Qualitäten zugleich verändert. Er senkt die Latenz (ein Redis-Zugriff im selben Rechenzentrum dauert typischerweise deutlich unter einer Millisekunde; eine Datenbankabfrage, die fünf Tabellen verknüpft, kann Dutzende Millisekunden dauern). Er erhöht den Durchsatz und schützt die Datenbank, weil jeder Treffer eine Abfrage ist, die die Datenbank nie sieht. Und er kann die Verfügbarkeit verbessern, weil ein Cache, der leicht veraltete Daten ausliefern darf, weiterarbeitet, während die Quelle langsam oder ausgefallen ist.

Der Preis wird mit Korrektheit und Komplexität bezahlt. Ein Cache ist eine zweite Kopie der Daten, und zwei Kopien können sich widersprechen. Jeder Cache verkörpert daher eine Entscheidung darüber, wie veraltet eine Antwort sein darf, und für wen. Eine fünf Minuten alte Speisekarte ist akzeptabel; ein fünf Minuten alter Kontostand, der einem Kunden angezeigt wird, der entscheidet, ob er sich eine Zahlung leisten kann, ist es nicht. Caches fügen außerdem eine Komponente hinzu, die ausfallen kann, verbrauchen Speicher, der Geld kostet, und schaffen neue Fehlerarten wie Stampedes und Kaltstarts, die es ohne sie nicht gäbe. Die erste Frage lautet daher nicht „Welcher Cache?“, sondern „Was darf veraltet sein, wie stark, und was passiert, wenn es das ist?“

Die Wirksamkeit eines Cache wird an seiner Trefferquote gemessen: Treffer geteilt durch alle Zugriffe. Die Trefferquote hängt davon ab, wie ungleich das Zugriffsmuster verteilt ist. Der Lesetraffic der meisten Endkundensysteme ist stark ungleich verteilt (ein kleiner Teil der Restaurants, Produkte oder Konten erhält die meisten Lesezugriffe), sodass ein Cache, der wenige Prozent der Daten hält, oft die meisten Lesezugriffe beantworten kann. Gleichmäßig zufällige Zugriffe dagegen lassen sich schlecht cachen, was man auch tut.

Die fünf klassischen Muster

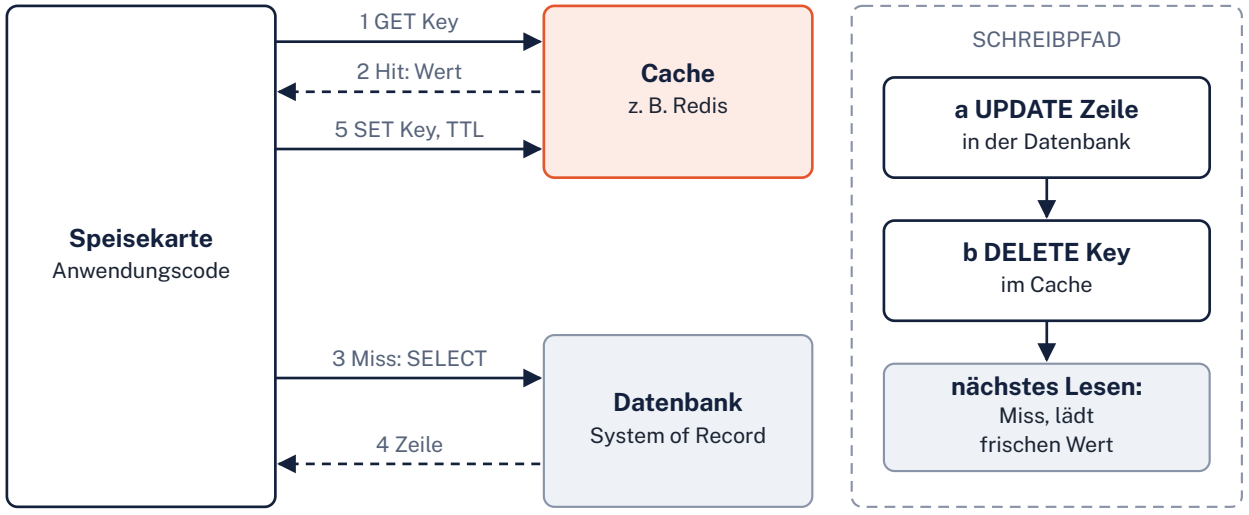
Caching-Muster unterscheiden sich darin, wer den Cache befüllt, wann Schreibvorgänge die Quelle erreichen und folglich wie veraltet und wie dauerhaft die Daten sein können. Die Tabelle vergleicht sie; die Absätze danach erklären, wo jedes passt.

Caching-Muster im Vergleich

Muster	Wie Lesen funktioniert	Wie Schreiben funktioniert	Stärken	Risiken
Cache-Aside (Lazy Loading)	Die Anwendung prüft den Cache; bei einem Fehltreffer liest sie die Datenbank und speichert das Ergebnis	Die Anwendung schreibt in die Datenbank und löscht (oder aktualisiert) dann den Cache-Eintrag	Einfach, cacht nur, was genutzt wird, ein Cache-Ausfall führt nur zu langsameren Lesezugriffen	Erster Lesezugriff nach Ablauf ist langsam; Race Conditions können veraltete Einträge hinterlassen; Stampedes
Read-Through	Die Anwendung fragt den Cache; die Cache-Bibliothek selbst lädt bei einem Fehltreffer aus der Datenbank	Meist kombiniert mit Write-Through oder Schreiben wie bei Cache-Aside	Die Ladelogik liegt an einer Stelle	Dieselben Probleme mit Veralten und Stampedes; die Cache-Schicht muss die Datenquelle kennen
Write-Through	Wie bei Read-Through oder Cache-Aside	Der Schreibvorgang geht synchron an Cache und Datenbank, bevor er bestätigt wird	Der Cache ist nie älter als die Datenbank, soweit Daten über ihn geschrieben werden	Jeder Schreibvorgang zahlt beide Latenzen; cacht Daten, die vielleicht nie gelesen werden
Write-Behind (Write-Back)	Aus dem Cache bedient	Der Schreibvorgang wird bestätigt, sobald er im Cache ist; später, oft gebündelt, in die Datenbank geschrieben	Sehr schnelle Schreibvorgänge; fängt Lastspitzen ab; bündelt Datenbankarbeit	Datenverlust, wenn der Cache vor dem Schreiben ausfällt; Datenbank hinkt hinterher; Reihenfolge und Konflikte
Refresh-Ahead	Aus dem Cache bedient	Betrifft keine Schreibvorgänge: heiße Einträge werden kurz vor ihrem Ablauf neu geladen	Beliebte Einträge laufen nie vor den Augen der Nutzer ab	Vergeudetes Neuladen von Einträgen, die nicht gelesen worden wären; braucht Zugriffsstatistiken

Cache-Aside ist aus gutem Grund der Standard: Es ist leicht nachvollziehbar und fällt sicher aus, denn verschwindet der Cache, liest die Anwendung einfach die Datenbank (langsamer). Read-Through verlagert dieselbe Logik in eine Cache-Bibliothek oder ein Cache-Produkt, was aufgeräumt ist, sich im Verhalten aber nicht unterscheidet. Write-Through passt zu Daten, die kurz nach dem Schreiben gelesen werden, etwa der Sitzung oder dem Warenkorb eines Nutzers. Write-Behind ist das riskante Muster: Es macht den Cache vorübergehend zum System of Record und ist daher nur angebracht, wo der Verlust der letzten paar Sekunden an Schreibvorgängen akzeptabel ist (Aufrufzähler, Analytics-Ereignisse, Spielstände) oder wo der Cache selbst repliziert und dauerhaft ist. Verwenden Sie Write-Behind nie für Geld. Refresh-Ahead ist eine Optimierung für eine kleine Menge sehr heißer Schlüssel, etwa die Restaurantliste auf der Startseite in einer belebten Stadt.

Cache-Aside in der Praxis



Lesezugriffe befüllen den Cache bei einem Fehltreffer nach Bedarf; Schreibvorgänge ändern zuerst die Datenbank und löschen dann die Kopie im Cache, statt sie zu überschreiben.

```
import json, random

TTL = 300  # seconds

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

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

Cache-Aside mit versioniertem Schlüssel, einer Sperre pro Schlüssel, um gleichzeitige Fehltreffer zusammenzufassen, gestreutem Ablauf und Invalidierung durch Löschen nach dem Schreiben in die Datenbank.

Time to Live und Invalidierung

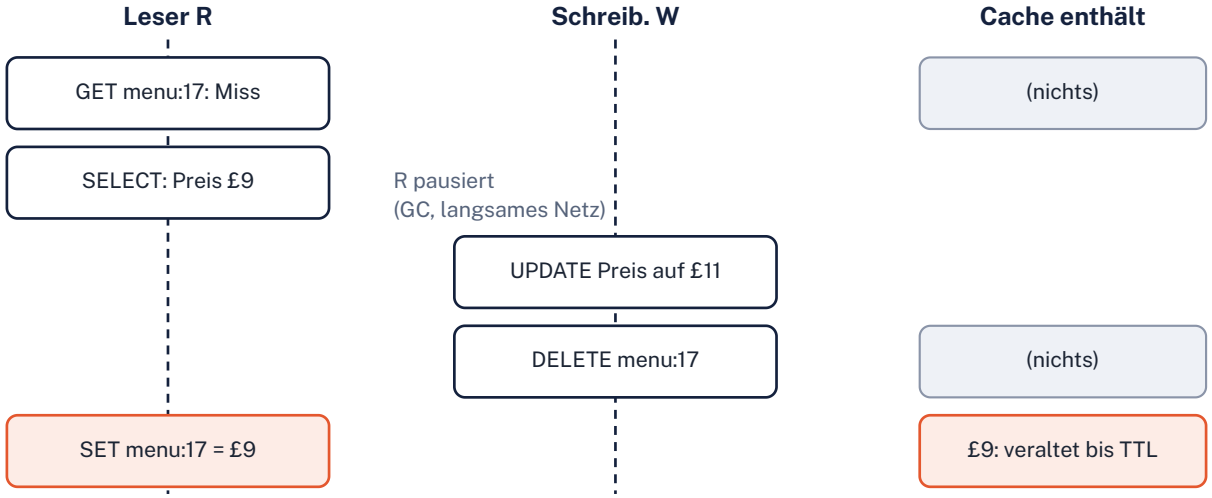
Es gibt zwei Wege, einen Cache daran zu hindern, geänderte Daten auszuliefern. Der erste ist eine Time to Live (TTL): Jeder Eintrag läuft nach einer festen Zeitspanne ab, was das Veralten ganz ohne Koordination begrenzt. Eine TTL von fünf Minuten für Speisekarten bedeutet, dass eine Preisänderung innerhalb von fünf Minuten überall sichtbar ist. Der zweite ist explizite Invalidierung: Ändert sich die Quelle, löscht oder aktualisiert etwas die Kopie im Cache. Invalidierung liefert frischere Daten, verlangt aber, dass jeder Codepfad, der die Daten ändert, jeden Cache kennt, der sie hält, und genau das geht schief, wenn Systeme wachsen. In verteilten Systemen wird Invalidierung oft durch Ereignisse ausgelöst: Der Speisekarten-Service veröffentlicht `MenuChanged`, und jeder Service, der eine Kopie hält, löscht seinen Eintrag. Die meisten professionellen Entwürfe kombinieren beides: Invalidierung bei Änderungen für Aktualität und eine TTL als Sicherheitsnetz für die Invalidierung, die verpasst wurde.

Die Schwierigkeit fasst ein Bonmot zusammen, das Phil Karlton von Netscape zugeschrieben wird: „In der Informatik gibt es nur zwei schwierige Dinge: Cache-Invalidierung und das Benennen von Dingen.“ Es ist schwierig, weil Invalidierung ein verkapptes Problem verteilter Systeme ist. Der Schreibvorgang in der Datenbank und das Löschen im Cache sind zwei

Operationen auf zwei Systemen ohne gemeinsame Transaktion; die eine kann gelingen, während die andere scheitert, und Operationen verschiedener Anfragen können sich überlappen.

Zwei Regeln vermeiden die meisten Probleme. Erstens: Aktualisieren Sie bei einem Schreibvorgang die Datenbank und **löschen** Sie dann den Cache-Eintrag; schreiben Sie nicht den neuen Wert in den Cache. Zwei gleichzeitige Schreiber, die jeweils den Cache setzen, können in umgekehrter Reihenfolge zu ihren Datenbank-Commits fertig werden und den älteren Wert auf unbestimmte Zeit im Cache hinterlassen, während ein Löschen höchstens einen zusätzlichen Fehltreffer verursacht. Zweitens: Akzeptieren Sie, dass selbst Löschen beim Schreiben eine schmale Race Condition übrig lässt, wie unten gezeigt, und begrenzen Sie sie mit einer TTL. Facebooks Paper von 2013 zur Skalierung von memcache beschreibt für diesen Fall **Leases**: Der Cache gibt dem Client, der einen Fehltreffer hatte, ein Token und lehnt dessen späteren Schreibvorgang ab, wenn der Schlüssel in der Zwischenzeit gelöscht wurde.

Die Stale-Set-Race-Condition



Löschen beim Schreiben verkleinert das Zeitfenster für veraltete Daten, schließt es aber nicht; eine TTL begrenzt, wie lange der veraltete Wert überlebt.

WENN DAS NEU FÜR SIE IST

Schritt für Schritt: warum löschen statt aktualisieren

Die Idee, die hier die meiste Sorgfalt verlangt, ist die Cache-Invalidierung: Die Datenbank zu ändern und den Cache zu ändern, sind getrennte Schritte auf zwei Systemen, sodass sich überlappende Anfragen einen alten Wert zurücklassen können. Zwei schnelle Adressänderungen zeigen, warum man löschen und nicht überschreiben sollte.

- Die Plattform cacht das Profil jedes Kunden, einschließlich Lieferadresse, mit Cache-Aside: Eine Anfrage mit Fehltreffer lädt das Profil aus der Datenbank und speichert es. Im Cache steht Wohnung 1. Innerhalb einer Sekunde speichert der Kunde Wohnung 2 und korrigiert dann auf Wohnung 3: Anfragen A und B.
- Angenommen, jedes Speichern schreibt zuerst in die Datenbank und überschreibt dann die Adresse im Cache. A schreibt Wohnung 2, B schreibt Wohnung 3: Die Datenbank stimmt.
- Doch A stockt zwischen seinen beiden Schritten. B legt Wohnung 3 in den Cache; dann legt der verspätete A Wohnung 2 hinein. Der Cache widerspricht nun der Datenbank, und jeder Leser sieht die falsche Wohnung.
- Nichts repariert das: Es kommt kein weiteres Speichern, also bleibt die falsche Adresse, bis der Eintrag abläuft, oder auf unbestimmte Zeit, wenn er nie abläuft.
- Lassen Sie nun jedes Speichern stattdessen das Profil aus dem Cache entfernen. In welcher Reihenfolge die Löschungen auch ankommen, am Ende ist der Cache leer; die nächste Anfrage hat einen Fehltreffer und speichert Wohnung 3 aus der Datenbank. Eine verspätete Löschung kostet höchstens einen zusätzlichen Fehltreffer.
- Entfernen verkleinert die Lücke, ohne sie zu schließen: Ein langsamer Leser kann direkt nach einer Löschung immer noch einen alten Wert speichern, wie die Race Condition oben zeigt. Eine Time to Live für jeden Eintrag begrenzt, wie lange das überlebt.
- **Die Lehre:** Schreiben Sie bei einer Änderung zuerst in die Datenbank, entfernen Sie dann die Kopie im Cache, statt sie zu ersetzen, und geben Sie jedem Eintrag eine Ablaufzeit, um aufzufangen, was die Invalidierung verpasst.

Stampedes und die Thundering Herd

Eine Cache Stampede entsteht, wenn ein beliebter Eintrag abläuft oder ein Cache-Knoten leer neu startet und Hunderte gleichzeitiger Anfragen im selben Augenblick einen Fehltreffer haben. Jede geht zur Datenbank, um denselben Wert zu berechnen, und die Datenbank, dimensioniert für die normale Fehltrefferrate des Cache, wird überlastet. Langsamere Antworten bedeuten, dass sich mehr Anfragen stauen, und die Überlast kann sich kaskadenartig ausbreiten. Die verwandte **Thundering Herd** beschreibt viele wartende Clients,

die gleichzeitig geweckt werden und um eine Ressource konkurrieren. Facebook berichtete, dass Leases bei einer Gruppe besonders anfälliger Schlüssel die Spitzenrate der Datenbankabfragen von 17.000 auf 1300 pro Sekunde senkten.

Die Abwehrmaßnahmen lassen sich gut kombinieren. Request Coalescing (auch Single Flight genannt) lässt nur eine Anfrage pro Schlüssel bei der Quelle abrufen, während die anderen auf ihr Ergebnis warten; innerhalb eines Prozesses ist das eine Map laufender Futures, prozessübergreifend eine kurze Sperre im Cache, wie im Code oben. **Gestreuter Ablauf (Jitter)** addiert zu jeder TTL einen zufälligen Betrag, damit gemeinsam geschriebene Einträge (nach einem Deployment oder einem Leeren des Cache) nicht gemeinsam ablaufen. **Stale-While-Revalidate** liefert allen den abgelaufenen Wert aus, während eine Hintergrundanfrage einen frischen holt, was die Latenzspitze vollständig beseitigt, wenn leichtes Veralten akzeptabel ist. **Probabilistischer vorzeitiger Ablauf** (Vattani, Chierichetti und Lowenstein, VLDB 2015) lässt jeden Leser zufällig, mit einer Wahrscheinlichkeit, die mit nahendem Ablauf steigt, über eine vorzeitige Aktualisierung entscheiden, sodass meist eine Anfrage aktualisiert, bevor der Eintrag tatsächlich abläuft. Schließlich vermeidet das **Aufwärmen** eines neuen Cache anhand einer Liste heißer Schlüssel, bevor er Traffic annimmt, die Kaltstart-Stampede nach einem Neustart.

AUSGEARBEITETES BEISPIEL

Ausgearbeitetes Beispiel: was die Trefferquote mit Latenz und Datenbanklast macht

Der Speisekarten-Service der Essenslieferplattform erhält zur abendlichen Spitze 20.000 Speisekarten-Lesezugriffe pro Sekunde. Ein Cache-Zugriff dauert 0,5 ms, eine Datenbankabfrage 8 ms. Bei einem Fehltreffer zahlt der Service beides (8,5 ms) und speichert das Ergebnis asynchron. Die Datenbank verkraftet etwa 3000 solcher Abfragen pro Sekunde, bevor die Latenz steil ansteigt.

Formel	Durchschnittliche Latenz = 0,5 + (1 − h) × 8 ms. Datenbanklast = (1 − h) × 20.000 Abfragen pro Sekunde.
Gar kein Cache	8 ms im Durchschnitt; 20.000 Abfragen pro Sekunde, fast das Siebenfache dessen, was die Datenbank verkraftet.
Trefferquote 80 %	0,5 + 0,2 × 8 = 2,1 ms; 4000 Abfragen pro Sekunde: immer noch über der Kapazität, also staut sich die Datenbank, und die Latenz steigt.
Trefferquote 95 %	0,5 + 0,05 × 8 = 0,9 ms; 1000 Abfragen pro Sekunde: komfortabel.
Trefferquote 99 %	0,5 + 0,01 × 8 = 0,58 ms; 200 Abfragen pro Sekunde.
Von 95 % auf 99 %	Die durchschnittliche Latenz verbessert sich nur um 0,32 ms, aber die Datenbanklast sinkt auf ein Fünftel, von 1000 auf 200 Abfragen pro Sekunde.
Cache-Cluster startet leer neu	Die Trefferquote fällt eine Weile auf nahezu 0 %: 20.000 Abfragen pro Sekunde treffen auf eine Datenbank, die für 3000 dimensioniert ist. Ohne Coalescing, Aufwärmen oder Rate Limiting bricht die Datenbank zusammen.
Perzentile	Bei 95 % Treffern dauern 5 % der Anfragen 8,5 ms, also liegt p95 genau auf der Kante, und p99 beträgt 8,5 ms, nicht 0,9 ms.

Die Trefferquote ist für die Datenbank weit wichtiger als für die durchschnittliche Latenz: Jede zusätzliche Neun an Treffern teilt die Datenbanklast etwa durch zehn.

Ein Cache, ohne den das System nicht laufen kann, ist keine Optimierung mehr; er ist eine kritische Abhängigkeit, und sein Ausfall muss eingeplant werden, so wie Abschnitt 2.5 jeden anderen einplant.

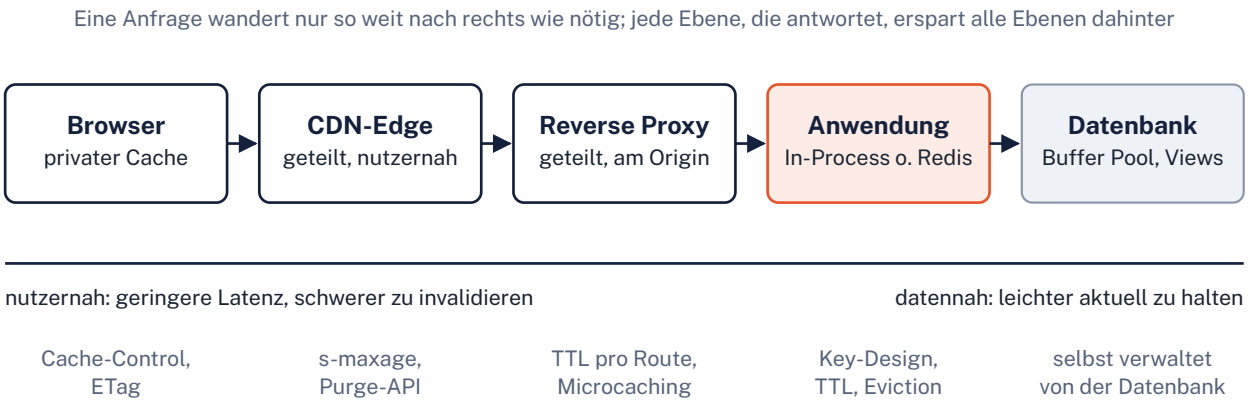
Durchschnittswerte schmeicheln Caches; wie Abschnitt 2.4 gezeigt hat, bestimmen die Fehltreffer die Tail-Latenz.

Caching-Ebenen

Caches gibt es auf jeder Ebene zwischen Nutzer und Festplatte, und eine Anfrage wird von der ersten Ebene beantwortet, die eine brauchbare Kopie hat. Der **Browser** führt einen privaten Cache, der über HTTP-Header gesteuert wird. Ein Content Delivery Network (CDN) hält geteilte Kopien an Edge-Standorten nahe bei den Nutzern, ideal für Bilder, Skripte und öffentliche API-Antworten wie Speisekarten. Ein **Reverse Proxy** am Origin (Nginx, Varnish) kann ganze Antworten für einige Sekunden cachen; das heißt Microcaching und kann eine plötzliche Lastspitze abfangen. Die **Anwendung** cacht berechnete Objekte, entweder prozessintern (am schnellsten, aber jede Instanz hält ihre eigene Kopie, und die Kopien widersprechen sich) oder in einem gemeinsamen Cache wie Redis oder Memcached. Die **Datenbank** cacht Seiten im Arbeitsspeicher (die Shared Buffers von PostgreSQL, der InnoDB-Buffer-Pool) und kann teure Ergebnisse in materialisierten Views vorberechnen.

Je weiter von den Daten entfernt, desto größer die Ersparnis pro Treffer und desto schwieriger die Invalidierung. Einen Schlüssel aus Redis können Sie in einer Millisekunde löschen; ein CDN können Sie leeren, langsamer; in den Browser eines Kunden können Sie gar nicht hineingreifen, also ist jedes `max-age`, das Sie senden, ein Versprechen, mit dem Sie leben müssen. Lange Browser-Lebensdauern werden daher für Dateien verwendet, deren URL sich ändert, wenn sich ihr Inhalt ändert (`app.3f9a1c.js`), und kurze, oder Revalidierung, für alles andere.

Wo Caches sitzen



Jede Ebene, die eine Anfrage beantwortet, erspart alle Ebenen dahinter; Ebenen näher am Nutzer sparen mehr, sind aber schwerer zu invalidieren.

HTTP-Caching-Header

HTTP hat ein vollständiges Caching-Modell, definiert in RFC 9111 (Juni 2022), und wer es gut nutzt, kann Last abfangen, bevor sie Ihre Server erreicht. `Cache-Control: max-age=60` erlaubt jedem Cache, eine Antwort 60 Sekunden lang wiederzuverwenden; `s-maxage` überschreibt das für geteilte Caches wie CDNs; `private` verbietet geteilten Caches, sie zu speichern

(verwenden Sie es für alles Persönliche); `no-cache` bedeutet verwirrenderweise „speichern, aber vor jeder Wiederverwendung beim Server nachfragen“; und `no-store` bedeutet „nie eine Kopie behalten“, die richtige Einstellung für Kontostände und Kontoauszüge. Die Erweiterungen `stale-while-revalidate` und `stale-if-error` (RFC 5861, 2010) wenden die obigen Ideen zu Stampedes und Verfügbarkeit auf HTTP-Ebene an.

Ist eine Kopie im Cache veraltet, muss der Client sie nicht erneut herunterladen. Ein Validator lässt ihn fragen, ob sich etwas geändert hat: Der Server sendet mit der Antwort ein `ETag` (eine undurchsichtige Versionskennung) oder ein `Last-Modified`-Datum, und der Client sendet später `If-None-Match` oder `If-Modified-Since`. Hat sich nichts geändert, antwortet der Server mit `304 Not Modified` ohne Body, was Bandbreite und oft den größten Teil der Serverarbeit spart. Diese bedingten Anfragen sind in RFC 9110 definiert. Der Header `Vary` teilt Caches mit, welche Anfrage-Header (etwa `Accept-Language`) unterschiedliche Antworten erzeugen, damit eine französische Speisekarte nie auf eine englische Anfrage ausgeliefert wird.

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

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

... 18 KB of menu JSON ...

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

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

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

Eine öffentliche Speisekarten-Antwort, die Browser 60 Sekunden und CDNs 300 Sekunden wiederverwenden dürfen, günstig revalidiert mit einem ETag; persönliche Finanzdaten verwenden no-store.

Verdrängung, Dimensionierung und Konsistenzrisiken

Ein Cache ist kleiner als die Daten dahinter, also muss er etwas verdrängen, wenn er voll ist. Least Recently Used (LRU) verdrängt den Eintrag, der am längsten unberührt ist, und funktioniert gut, wenn jüngste Nutzung künftige Nutzung vorhersagt. Least Frequently Used (LFU) verdrängt den am seltensten genutzten Eintrag und widersteht einmaligen Scans (ein nächtlicher Bericht, der jedes Restaurant einmal berührt, würde einen LRU-Cache seiner wirklich heißen Einträge entleeren). Produktive Caches nähern beides an: Redis etwa zieht eine Stichprobe weniger Schlüssel, verdrängt den besten Kandidaten darunter und bietet Strategien wie `allkeys-lru`, `allkeys-lfu` und `volatile-ttl`. Seine Standardstrategie `noeviction` lehnt Schreibvorgänge ab, wenn der Speicher voll ist, was zu Redis als Datenspeicher passt, aber nicht als Cache; legen Sie die Strategie also bewusst fest.

Über das Veralten hinaus bringen Caches subtilere Konsistenzrisiken mit sich. Ein Nutzer, der seine Lieferadresse ändert und dann die alte sieht, weil der Lesezugriff aus einem Cache kam, den eine andere Instanz noch nicht invalidiert hatte, erlebt eine gebrochene Read-your-Writes-Garantie (Abschnitt 7.4); eine gängige Lösung ist, den Cache für die eigenen jüngsten Schreibvorgänge eines Nutzers zu umgehen. Prozessinterne Caches auf vielen Instanzen widersprechen einander, sodass aufeinanderfolgende Anfragen, die an verschiedene Instanzen gehen, einen Wert scheinbar rückwärts laufen lassen. Einen Fehler oder ein leeres Ergebnis ohne kurze TTL zu cachen, macht aus einem kurzen Ausfall einen langen. Daten zu cachen, die Nutzer vermischen (eine Seite mit dem Namen des Kunden, nur unter der URL gecacht), gibt personenbezogene Daten an den nächsten Besucher preis, eine bekannte Klasse von Sicherheitsvorfällen. Und Cache-Schlüssel müssen alles enthalten, was die Antwort verändert: Mandant, Sprache, Währung, Berechtigungen und die Version des Datenformats.

ACHTUNG

Häufige Fehler

- Einen Cache einführen, bevor gemessen wurde: Oft ist die eigentliche Lösung ein Index oder eine bessere Abfrage, die das Problem ohne zweite Kopie der Daten beseitigt.
- Gar keine TTL, sodass man sich ganz auf Invalidierung verlässt, die ein vergessener Codepfad verpassen wird.
- Den Cache beim Schreiben aktualisieren statt ihn zu löschen, was Reihenfolge-Races zwischen Schreibern erzeugt.
- Die Datenbank für die gecachte Last dimensionieren, sodass ein Cache-Neustart die ganze Plattform lahmlegt.
- Personenbezogene oder finanzielle Daten in geteilten Caches ablegen oder `private` und `Vary` bei HTTP-Antworten vergessen.
- Trefferquote, Verdrängungsrate und Speicher im Monitoring ignorieren, sodass ein Cache unbemerkt aufhört zu helfen.

PRÜFEN SIE IHR VERSTÄNDNIS

▼ Warum sollte ein Schreibvorgang den Cache-Eintrag löschen, statt den neuen Wert in den Cache zu schreiben?

Weil zwei gleichzeitige Schreiber ihre Cache-Aktualisierungen in umgekehrter Reihenfolge zu ihren Datenbank-Commits abschließen können, sodass der ältere Wert im Cache bleibt; ein Löschen verursacht schlimmstenfalls einen zusätzlichen Fehltreffer, der den aktuellen Wert neu lädt.

▼ Ein Cache-Zugriff kostet 1 ms und ein Datenbank-Lesezugriff 10 ms. Wie hoch ist ungefähr die durchschnittliche Latenz bei einer Trefferquote von 90 % mit Cache-Aside?

1 + 0,1 × 10 = 2 ms (jede Anfrage zahlt den Cache-Zugriff; eine von zehn zahlt zusätzlich den Datenbank-Lesezugriff).

▼ Was ist der Unterschied zwischen Cache-Control: no-cache und no-store?

no-cache erlaubt einem Cache, die Antwort zu speichern, verlangt aber, dass er sie vor jeder Wiederverwendung beim Server revalidiert; no-store verbietet überhaupt, eine Kopie zu behalten.

▼ **Warum kann es wichtig sein, die Trefferquote von 99 % auf 99,9 % zu erhöhen, obwohl sich die durchschnittliche Latenz kaum ändert?**

Weil die Datenbanklast proportional zur Fehltrefferrate ist, die von 1 % auf 0,1 % fällt, also zehnmal weniger Abfragen, die die Datenbank erreichen.

▼ **Probieren Sie es aus: Ein Speisekarten-Service erhält 12.000 Lesevorgänge pro Sekunde. Ein Cache-Lookup dauert 0,5 ms, eine Datenbankabfrage 8 ms, und die Datenbank bewältigt 1000 Abfragen pro Sekunde. Welche Trefferquote brauchen Sie? Was passiert, wenn der Cache leer neu startet?**

Fehlschläge dürfen höchstens 1000 von 12.000 sein, die Trefferquote muss also mindestens 91,7 % betragen. Bei 95 % sieht die Datenbank 600 Abfragen pro Sekunde, und die durchschnittliche Latenz beträgt 0,5 + 0,05 x 8 = 0,9 ms. Ein kalter Cache schickt alle 12.000 an die Datenbank, das Zwölffache ihres Limits, was sie zusammenbrechen lassen kann. Wärmen Sie den Cache auf, bevor er Verkehr annimmt, fassen Sie Anfragen für denselben Schlüssel zusammen, und versehen Sie Ablaufzeiten mit Jitter.

QUELLEN

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

Schemaevolution, Migrationen und Kompatibilität

BEVOR SIE DAS LESEN

Dieser Abschnitt baut auf:

- 0.5 Daten speichern
- 0.8 Wie Software entsteht
- 6.5 Kommunikation und APIs

Begriffe von früher, die er verwendet: Breaking Change, Protocol Buffers, Continuous Delivery, Replikationsverzögerung (Replication Lag), Idempotent

DIE IDEE IN KÜRZE

Stellen Sie sich ein Krankenhaus vor, das das Papierformular für jede Aufnahme ändert, während weiter Patientinnen und Patienten eintreffen. Tauscht es die Formulare einfach über Nacht aus, hält die Nachtschicht noch das alte Formular in der Hand, während die Tagschicht das neue erwartet, und die abgelegten Akten nutzen weiterhin das alte Layout. Ein umsichtiges Krankenhaus druckt ein Formular mit den alten und den neuen Feldern, verwendet es, bis alle umgestellt haben und die alten Akten übertragen sind, und entfernt erst dann die alten Felder. Softwareteams ändern die Struktur ihrer Daten genau auf diese umsichtige, schrittweise Art.

WAS SIE WISSEN MÜSSEN

- Schemas müssen sich ändern, während alter und neuer Code nebeneinander laufen und alte Daten noch existieren; deshalb muss jeder deployte Schemazustand mit mindestens zwei Codeversionen funktionieren.
- Abwärtskompatibilität bedeutet, dass neue Leser alte Daten lesen können; Aufwärtskompatibilität bedeutet, dass alte Leser neue Daten lesen können; volle Kompatibilität erlaubt es, Leser und Schreiber in beliebiger Reihenfolge zu aktualisieren.
- Expand and Contract (Parallel Change) zerlegt eine inkompatible Änderung in kompatible Schritte: die neue Struktur hinzufügen, doppelt schreiben und Backfill durchführen, die Leser umstellen, das Schreiben der alten Struktur beenden und sie dann entfernen.
- Migrationen sind versionierte Skripte, die beim Code liegen und von Werkzeugen wie Flyway, Liquibase und Alembic automatisch angewendet werden; bearbeiten Sie nie eine bereits angewendete Migration, und proben Sie mit Daten in Produktionsgröße.

- Große Backfills laufen als separate, in Batches aufgeteilte, gedrosselte und idempotente Jobs; Online-Techniken wie nebenläufige Indexerstellung und NOT-VALID-Constraints vermeiden lange Sperren.
- Schema Registries erzwingen Kompatibilitätsregeln für Event-Schemas; Avro setzt auf Standardwerte und die Auflösung zwischen Schreiber- und Leserschema, Protocol Buffers darauf, Feldnummern nie wiederzuverwenden.

Warum sich Schemas weiterentwickeln müssen, ohne die Welt anzuhalten

Alle gespeicherten oder übertragenen Daten haben ein Schema: die Tabellen und Spaltentypen einer Datenbank, die Felder einer JSON-API-Antwort, die Datensatzdefinition eines Events in einem Stream. Schemas ändern sich ständig, weil sich das Geschäft ändert: Die Bank führt einen zweiten Kontoinhaber ein, die Essenslieferplattform beginnt, Trinkgelder zu erfassen, eine Aufsichtsbehörde verlangt ein neues Feld. Die architektonische Frage lautet, wie man ein Schema ändert, während das System weiterläuft, obwohl die alte Form noch von Code, gespeicherten Daten und anderen Teams verwendet wird.

Drei Tatsachen machen das schwierig. Erstens laufen in jedem System, das ohne Ausfallzeit deployt wird, alte und neue Codeversionen nebeneinander: Ein Rolling Deployment ersetzt Instanzen nacheinander, ein Canary leitet einen kleinen Teil des Datenverkehrs an die neue Version, und ein Rollback holt die alte Version zurück. Jeder Schemazustand muss daher mit mindestens zwei Codeversionen funktionieren. Zweitens leben Daten länger als Code. Vor fünf Jahren geschriebene Zeilen, dauerhaft aufbewahrte Events in einem Log, in einer Queue wartende Nachrichten und Backups behalten alle die alte Form. Drittens werden Schemas gemeinsam genutzt: Eine Spalte wird von Berichten, Extraktionsjobs und manchmal anderen Services gelesen, und ein Event-Schema wird von Consumern gelesen, die der Producer vielleicht gar nicht kennt. Abschnitt 7.1 hat dafür plädiert, Daten hinter einer Schnittstelle zu besitzen; bei der Schemaevolution zahlt sich diese Disziplin aus, denn ein privates Schema lässt sich mit der Zustimmung eines einzigen Teams ändern.

Abwärts-, Aufwärts- und volle Kompatibilität

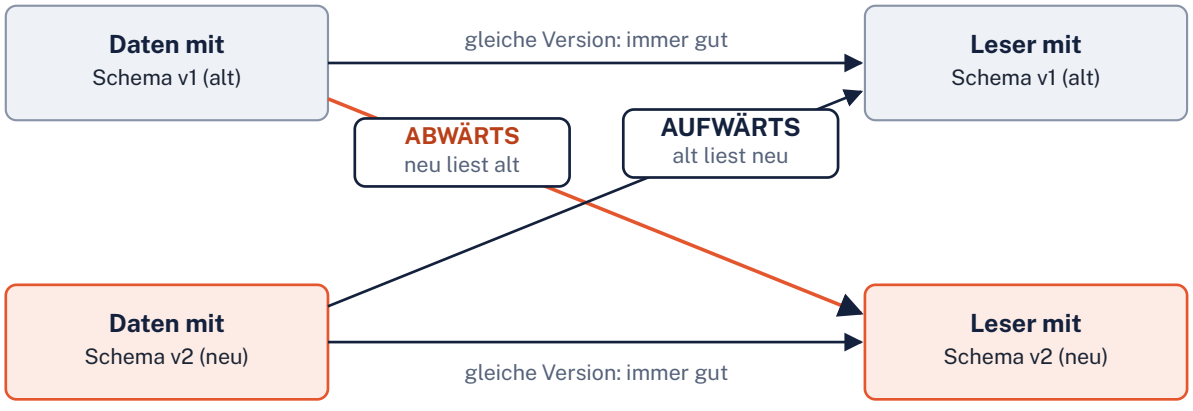
Kompatibilität wird über Leser und Schreiber definiert. Eine Änderung ist abwärtskompatibel, wenn Code mit dem **neuen** Schema Daten lesen kann, die mit dem **alten** geschrieben wurden. Ein optionales Feld mit Standardwert hinzuzufügen ist abwärtskompatibel: Der neue Leser setzt den Standardwert ein, wenn den alten Daten das Feld fehlt. Eine Änderung ist aufwärtskompatibel, wenn Code mit dem **alten** Schema Daten lesen kann, die mit dem **neuen**

geschrieben wurden. Das setzt voraus, dass alte Leser Felder ignorieren, die sie nicht kennen; das ist die Disziplin des Tolerant Reader. Eine Änderung ist **voll kompatibel**, wenn sie beides ist; dann lassen sich Leser und Schreiber in beliebiger Reihenfolge aktualisieren.

Welche Richtung Sie brauchen, ergibt sich daraus, wer zuerst aktualisiert. Aktualisieren Consumer vor Producern (häufig, wenn ein Stream wenige Consumer hat, die Sie kontrollieren), brauchen Sie Abwärtskompatibilität. Aktualisieren Producer zuerst, was der übliche Fall bei einer Datenbank ist (neuer Code beginnt zu schreiben, bevor jeder Leser neu deployt wurde) und bei öffentlichen APIs, brauchen Sie Aufwärtskompatibilität. Wenn Sie die Reihenfolge nicht kontrollieren können oder beide Seiten zurückrollen müssen, streben Sie volle Kompatibilität an. Die Confluent Schema Registry prüft zum Beispiel standardmäßig Abwärtskompatibilität gegen die vorherige Version und bietet transitive Varianten, die gegen jede frühere Version prüfen; das ist wichtig, wenn Consumer alte Daten vom Anfang eines Logs erneut abspielen können.

Ein Wort der Warnung zur Begrifflichkeit. Im API-Design wird „abwärtskompatibel“ oft locker im Sinne von „bestehende Clients funktionieren weiter“ verwendet; in Begriffen von Lesern und Schreibern ist das Aufwärtskompatibilität für die Clients, die Ihre Antworten lesen, und Abwärtskompatibilität für die Anfragen, die Sie von ihnen lesen. Wenn eine Diskussion an dem Begriff hängt, fragen Sie, welche Seite sich ändert und welche Seite weiter funktionieren muss.

Welcher Leser welche Daten lesen kann



VOLL = beide Diagonalen funktionieren; Leser und Schreiber beliebig aktualisierbar

Abwärtskompatibilität schützt neue Leser vor alten Daten; Aufwärtskompatibilität schützt alte Leser vor neuen Daten.

Häufige Änderungen und ihre Kompatibilität

Änderung	Abwärts (neu liest alt)	Aufwärts (alt liest neu)	Sicheres Vorgehen
Ein optionales Feld oder eine nullable Spalte mit Standardwert hinzufügen	Ja	Ja, wenn Leser unbekannte Felder ignorieren	Die übliche sichere Änderung
Ein Pflichtfeld ohne Standardwert hinzufügen	Nein: Alten Daten fehlt es	Ja	Optional hinzufügen, Backfill, dann erzwingen
Ein Feld entfernen	Ja	Nur wenn alte Leser einen Standardwert dafür haben	Erst überall nicht mehr lesen, dann entfernen
Ein Feld oder eine Spalte umbenennen	Nein	Nein	Expand and Contract: neu hinzufügen, kopieren, umstellen, alt entfernen
Einen Typ ändern (Text zu Integer, int zu long)	Hängt vom Format ab: Erweitern ist oft sicher, Einengen nicht	Hängt davon ab	Ein neues Feld mit dem neuen Typ hinzufügen und migrieren
Bedeutung ändern, ohne die Form zu ändern (Pence zu Pfund)	Unbemerkt falsch	Unbemerkt falsch	Niemals: ein neues Feld mit neuem Namen anlegen

WENN DAS NEU FÜR SIE IST

Schritt für Schritt: alte Leser, neue Leser

Der Punkt, an dem hier die meisten stolpern, ist der Unterschied zwischen Abwärts- und Aufwärtskompatibilität und warum es davon abhängt, welche Seite zuerst aktualisiert, welche Sie brauchen. Verfolgen Sie ein neues Feld durch die Essenslieferplattform.

- Die Plattform beginnt, Trinkgelder zu erfassen; deshalb fügt eine neue Version des OrderPlaced-Events ein Feld hinzu, tip_pence. Bereits geschriebene und gespeicherte Events haben dieses Feld nicht.
- Angenommen, zuerst wird ein Consumer aktualisiert. Sein neuer Code liest Events, die geschrieben wurden, bevor es Trinkgelder gab, auch alte, die vom Anfang des Logs erneut abgespielt werden; also muss er das fehlende Feld mit einem Standardwert füllen, hier null, was bedeutet: kein Trinkgeld erfasst. Dieser Fall, in dem die neue Version mit alten Daten zurechtkommt, heißt Abwärtskompatibilität.
- Angenommen, stattdessen wird zuerst der Producer aktualisiert. Consumer, die noch alten Code ausführen, erhalten nun Events mit einem Feld, von dem sie nie gehört haben, und müssen es einfach ignorieren, wie es ein Tolerant Reader tut. Dieser Fall, in dem die alte Version mit neuen Daten zurechtkommt, heißt Aufwärtskompatibilität.
- Mit Tolerant Readern funktioniert ein optionales Feld mit Standardwert in beide Richtungen; die Änderung ist also voll kompatibel: Jede Seite kann zuerst aktualisieren, und jede kann zurückgerollt werden.
- Im Gegensatz dazu das Umbenennen von total_pence in amount_pence, für Leser, die Felder über den Namen finden. Neue Leser suchen amount_pence und finden es in alten Events nicht; alte Leser suchen total_pence und finden es in neuen nicht. Die Änderung bricht in beiden Richtungen zugleich.
- **Die Lehre:** Fragen Sie vor einer Schemaänderung, wer neue Daten mit altem Code lesen wird und wer alte Daten mit neuem Code. Additive Änderungen mit Standardwerten sind in beide Richtungen sicher; alles andere braucht die schrittweise Technik, die als Nächstes kommt.

Expand and Contract: die zentrale Technik

Die meisten inkompatiblen Änderungen lassen sich in eine Folge kompatibler Änderungen zerlegen. Das Muster heißt Expand and Contract oder Parallel Change (Danilo Satos Beschreibung von 2014 auf martinfowler.com nennt die Phasen Expand, Migrate und Contract). Zuerst **erweitern** (Expand): die neue Struktur neben der alten hinzufügen, sodass alter und neuer Code funktionieren. Dann **migrieren** (Migrate): die Schreiber so ändern, dass sie beide pflegen, vorhandene Daten übertragen und die Leser nacheinander auf die neue Struktur

umstellen. Schließlich **zurückbauen** (Contract): Sobald nichts mehr die alte Struktur nutzt, wird sie entfernt. Jeder Schritt wird separat deployt, jeder ist mit der Version davor kompatibel, und jeder lässt sich ohne Datenverlust zurückrollen.

Der Preis sind Zeit und vorübergehende Unordnung: Die Änderung erstreckt sich über mehrere Releases, und tage- oder wochenlang trägt das Schema beide Formen. Der Nutzen: Kein Release braucht jemals eine Ausfallzeit, kein Deployment muss teamübergreifend koordiniert werden, und jeder Schritt lässt sich anhalten. Scott Amblers und Pramod Sadalages *Refactoring Databases* (2006) katalogisiert Dutzende Datenbank-Refactorings, die auf derselben Idee beruhen, mit einer **Übergangsphase**, in der altes und neues Schema nebeneinander bestehen und von der Anwendung oder von Triggern synchron gehalten werden.

AUSGEARBEITETES BEISPIEL

Ausgearbeitetes Beispiel: eine Spalte ohne Ausfallzeit umbenennen

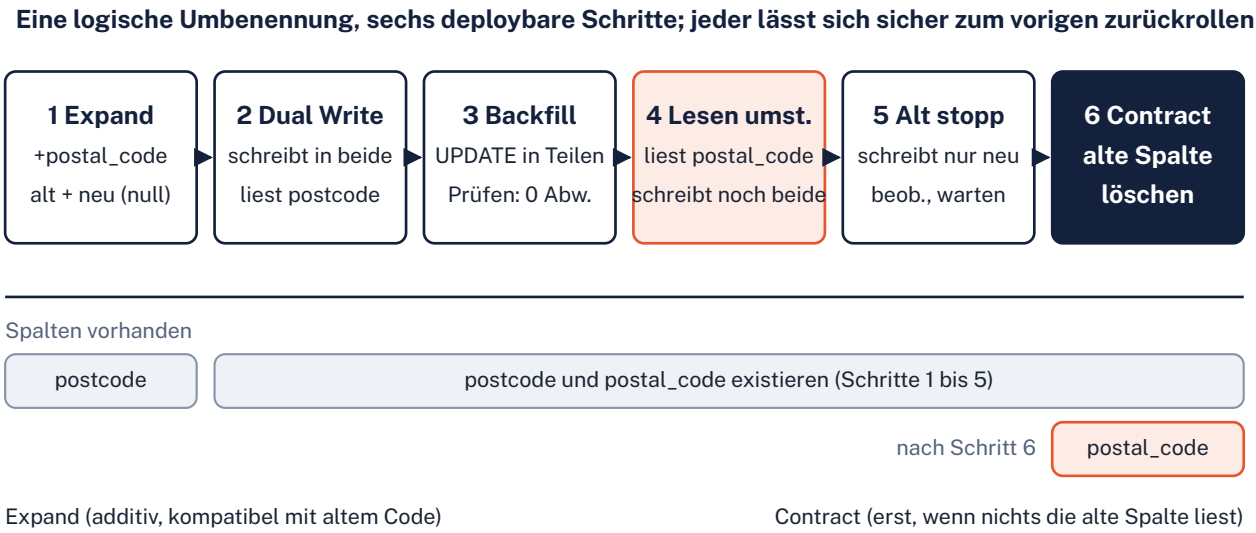
Die Tabelle restaurants der Essenslieferplattform hat eine Spalte `postcode`. Die Plattform expandiert in Länder mit anderen Postleitzahlformaten, und das Team möchte sie `postal_code` nennen. Die Tabelle hat 4 Millionen Zeilen, der Restaurant-Service wird mehrmals täglich mit Rolling Updates deployt, und ein nächtlicher Extraktionsjob liest die Spalte ebenfalls. Ein einzelnes `ALTER TABLE ... RENAME COLUMN` würde jede laufende Instanz des alten Codes in dem Moment brechen, in dem es committet wird.

Release 1: Expand	Eine Migration fügt `postal_code text NULL` hinzu. Ab PostgreSQL 11 ist das Hinzufügen einer Spalte ohne Standardwert oder mit konstantem Standardwert eine reine Metadatenänderung. Alter Code ignoriert die neue Spalte; nichts bricht.
Release 2: Dual Write	Die Anwendung schreibt bei jedem Insert und Update denselben Wert in beide Spalten und liest weiterhin `postcode`. (Ein Datenbank-Trigger, der eine Spalte in die andere kopiert, ist eine Alternative, wenn mehrere Anwendungen in die Tabelle schreiben.)
Schritt 3: Backfill	Ein Hintergrundjob kopiert `postcode` für bestehende Zeilen in Batches von 5000 nach Primärschlüsselbereich in `postal_code`, pausiert zwischen den Batches und beobachtet die Replikationsverzögerung. Danach zählt eine Prüfabfrage die Zeilen, in denen sich beide unterscheiden: Sie muss 0 liefern.
Release 4: Lesen umstellen	Die Anwendung liest `postal_code`, schreibt aber weiterhin beide Spalten, sodass ein Rollback auf Release 2 oder 3 sicher bleibt. Der Extraktionsjob wird ebenfalls auf die neue Spalte umgestellt.
Release 5: die alte Spalte nicht mehr schreiben	Die Anwendung schreibt nur noch `postal_code`. Die alte Spalte darf jetzt nicht `NOT NULL` sein, sonst schlagen Inserts fehl. Warten Sie lange genug, um sicher zu sein, dass kein Rollback auf Release 4 nötig wird.
Release 6: Contract	Eine Migration erzwingt `NOT NULL` für `postal_code` (mit einem `NOT VALID`-Check-Constraint, der separat validiert wird und den PostgreSQL ab Version 12 nutzen kann, um `NOT NULL` ohne langen blockierenden Scan zu setzen) und löscht schließlich `postcode`.

Sechs Deployments für eine Umbenennung klingen übertrieben, aber Schritte lassen sich zusammenlegen, wenn das Team jeden Leser kontrolliert und ein kurzes Rollback-Fenster akzeptiert; entscheidend ist das Prinzip, dass jeder deployte Zustand mit dem Code davor und danach funktioniert.

Den Contract-Schritt vergessen Teams am häufigsten. Verfolgen Sie ihn ausdrücklich (ein Ticket, ein ADR-Status, eine Lint-Regel, die die alte Spalte markiert), sonst füllt sich das Schema mit halb migrierten Spalten.

Die Umbenennung als sechs deploybare Schritte



In den Schritten 1 bis 5 existieren beide Spalten, sodass jede Codeversion aus dem vorigen Schritt weiterhin laufen kann.

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

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

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

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

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

PostgreSQL-Anweisungen für die Schritte Expand, Backfill, Prüfen und Contract, so geschrieben, dass lange Sperren vermieden werden.

Migrationen als Code in der Delivery-Pipeline

Datenbankänderungen gehören in die Versionsverwaltung neben den Anwendungscode, als geordnete Folge von Migrationen, jede ein kleines Skript, das das Schema von einer Version zur nächsten bringt. Pramod Sadalages und Martin Fowlers *Evolutionary Database Design* (erstmals 2003 veröffentlicht, 2016 neu geschrieben) hat die Praxis beschrieben, und die Werkzeuge sind ausgereift. **Flyway** führt versionierte SQL-Dateien wie `V41__add_postal_code.sql` der Reihe nach aus und verzeichnet jede in einer Tabelle `flyway_schema_history`. **Liquibase** beschreibt Änderungen als Changesets in SQL, XML, YAML oder JSON, erfasst in einer Tabelle `DATABASECHANGELOG`. **Alembic** für Python und SQLAlchemy erzeugt Revisionsskripte mit den Funktionen `upgrade()` und `downgrade()` und hält die aktuelle Revision in `alembic_version` fest. Flyway und Liquibase speichern außerdem eine Prüfsumme jeder angewendeten Migration und brechen mit einem Fehler ab, wenn eine davon nachträglich bearbeitet wurde; das schützt Umgebungen davor, unbemerkt auseinanderzulaufen.

In einer Continuous-Delivery-Pipeline laufen dieselben Migrationen in Tests gegen eine frische Datenbank, dann gegen Staging, dann gegen die Produktion, sodass die Produktion nie eine Änderung erhält, die nicht geprobt wurde. Gute Praxis ergänzt einige Regeln. Bearbeiten Sie

nie eine Migration, die irgendwo angewendet wurde; schreiben Sie eine neue. Halten Sie Schemaänderungen und den davon abhängigen Code in getrennten Deployments, bei Erweiterungen die Schemaänderung zuerst, bei Rückbauten zuletzt. Testen Sie Migrationen gegen eine Kopie von Daten in Produktionsgröße, denn eine Anweisung, die auf einer Testdatenbank 50 ms dauert, kann eine große Tabelle eine Stunde lang sperren. Rollen Sie lieber mit einer neuen Migration vorwärts, statt Down-Migrationen zu nutzen, die Spalten und damit Daten löschen. Und lassen Sie sich von der Datenbank über Sperren informieren: In PostgreSQL blockiert ein `ALTER TABLE`, das auf seine Sperre wartet, jede danach eintreffende Abfrage; setzen Sie also ein kurzes `lock_timeout` und versuchen Sie es erneut, statt sich hinter einem lang laufenden Bericht einzureihen.

Große Backfills und Online-Schemaänderungen

Ein Backfill füllt eine neue Spalte oder Tabelle für bestehende Zeilen. Als ein einziges `UPDATE` über 400 Millionen Transaktionen hält er stundenlang Sperren, erzeugt riesige Mengen Write-Ahead-Log, die Replikate nachspielen müssen (was Replikationsverzögerung verursacht), und hinterlässt in PostgreSQL tote Zeilenversionen, die Vacuum aufräumen muss. Der professionelle Ansatz ist ein separater, neu startbarer Job, der kleine Batches nach Primärschlüsselbereich verarbeitet, jeden Batch committet, seinen Fortschritt festhält, sich drosselt, wenn Replikationsverzögerung oder Datenbanklast steigen, und idempotent ist, sodass ein erneut ausgeführter Batch keinen Schaden anrichtet. Führen Sie ihn außerhalb des Migrationswerkzeugs aus, damit ein Deployment nie auf ihn wartet.

Manche Änderungen lassen sich nicht ohne Sperren an Ort und Stelle vornehmen. PostgreSQL bietet `CREATE INDEX CONCURRENTLY` (das nicht innerhalb eines Transaktionsblocks laufen kann, was man Migrationswerkzeugen mitteilen muss) und die oben gezeigte Abfolge `NOT VALID`, dann `VALIDATE CONSTRAINT`. Für MySQL bauen Werkzeuge wie gh-ost und pt-online-schema-change eine Schattenkopie der Tabelle mit dem neuen Schema, halten sie mit laufenden Schreibvorgängen synchron und tauschen sie mit einer kurzen Umbenennung ein. Den Typ einer Spalte ändert man meist am besten per Expand and Contract auf eine neue Spalte statt mit einem `ALTER` an Ort und Stelle, das in den meisten Datenbanken die ganze Tabelle neu schreibt.

Schema Registries, Avro und Protocol Buffers

Für Events und Nachrichten entspricht dem Migrationswerkzeug eine Schema Registry. Producer registrieren jede Schemaversion; die Registry prüft sie gegen die konfigurierte Kompatibilitätsregel und weist eine inkompatible Änderung beim Build oder Deployment zurück statt erst in der Produktion. Jede Nachricht trägt eine kurze Schemakennung, sodass Consumer genau das Schema abrufen können, mit dem sie geschrieben wurde.

Apache Avro macht das Schema des Schreibers explizit: Zum Dekodieren braucht man sowohl das Schema des Schreibers als auch das des Lesers, und Avros Auflösungsregeln ordnen Felder über den Namen zu, ignorieren Felder des Schreibers, die der Leser nicht kennt, und füllen Leserfelder, die in den Daten fehlen, mit ihren deklarierten Standardwerten. Daraus folgt die praktische Regel: Geben Sie jedem Feld, das Sie hinzufügen oder entfernen könnten, einen Standardwert. Im Beispiel unten wurden `currency` und `tip_pence` in späteren Versionen mit Standardwerten hinzugefügt, sodass alte Events mit neuen Consumern weiterhin dekodiert werden.

Protocol Buffers identifizieren Felder auf dem Übertragungsweg über ihre Nummer, nicht über den Namen. Felder hinzuzufügen ist sicher, und alte Leser behalten unbekannte Felder beim erneuten Serialisieren. Das Umbenennen eines Felds wirkt sich nicht auf die binäre Kodierung aus (wohl aber auf die JSON-Form). Die obersten Regeln lauten, eine Feldnummer nie zu ändern oder wiederzuverwenden und die Nummern und Namen gelöschter Felder als `reserved` zu markieren, damit niemand sie versehentlich wiederverwendet. Confluent empfiehlt für Protobuf rückwärts-transitive Kompatibilität (backward transitive), weil das Hinzufügen eines neuen Nachrichtentyps nicht aufwärtskompatibel ist. Auch JSON Schema lässt sich registrieren und prüfen, aber die Offenheit von JSON (ist eine unbekannte Eigenschaft erlaubt oder nicht?) macht die Kompatibilitätsregeln subtiler.

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

Ein Avro-Schema für ein OrderPlaced-Event; nach Version 1 hinzugefügte Felder haben Standardwerte, sodass neue Consumer alte Events lesen können.

API- und Event-Schemas weiterentwickeln

APIs und Events folgen denselben Prinzipien mit anderen Werkzeugen. Bevorzugen Sie bei APIs additive Änderungen, verlangen Sie von Clients, dass sie Tolerant Reader sind, und heben Sie inkompatible Änderungen für eine neue Version auf, die neben der alten läuft, mit Nutzungsmonitoring, um zu sehen, wann die alte Version abgeschaltet werden kann (Abschnitt

6.5 behandelt Versionierungsansätze). Mit Consumer-Driven-Contract-Tests veröffentlicht jeder Consumer die Felder, auf die er sich tatsächlich verlässt, sodass der Provider weiß, welche Änderungen sicher sind.

Events sind schwieriger als APIs, weil sie gespeichert werden: Ein Consumer kann ein Topic von Anfang an erneut abspielen, und ein System mit Event-Sourcing (Abschnitt 7.5) baut seinen Zustand aus Events auf, die vor Jahren geschrieben wurden. Behandeln Sie ein veröffentlichtes Event-Schema als öffentlichen Vertrag, versionieren Sie es in der Registry, und wenn eine Änderung wirklich inkompatibel ist, veröffentlichen Sie für eine gewisse Zeit einen neuen Event-Typ oder ein neues Topic (`OrderPlaced.v2`) neben dem alten, statt das bestehende zu verändern. Abschnitt 6.4 behandelt das Event-Design allgemeiner.

Checkliste: bevor Sie eine Schemaänderung ausliefern

- [ ] **Ist die Änderung additiv?**
  Wenn nicht, zerlegen Sie sie in Expand-, Migrate- und Contract-Schritte, die jeweils für sich deploybar sind.
- [ ] **Funktioniert jeder deployte Zustand mit der vorigen und der nächsten Codeversion?**
  Wegen Rolling Deployments und Rollbacks laufen immer zwei Versionen gleichzeitig.
- [ ] **Wer liest diese Daten sonst noch?**
  Suchen Sie nach Berichten, Extraktionsjobs, anderen Services und Event-Consumern; prüfen Sie die Abonnenten in der Registry und die Lineage im Datenkatalog.
- [ ] **Wurde die Migration mit Daten in Produktionsgröße ausgeführt?**
  Messen Sie Sperrdauer, Tabellen-Rewrites und Log-Volumen; nutzen Sie, wo verfügbar, nebenläufige Indexerstellung und NOT-VALID-Constraints.
- [ ] **Ist ein Lock-Timeout gesetzt?**
  Schnell scheitern und erneut versuchen, statt ein wartendes ALTER den gesamten Verkehr blockieren zu lassen.
- [ ] **Ist der Backfill separat, in Batches, gedrosselt, idempotent und fortsetzbar?**
  Er sollte nie ein Deployment blockieren oder Replikate überfluten.
- [ ] **Haben neue Felder Standardwerte, und werden keine Feldnummern oder -namen wiederverwendet?**
  Unverzichtbar für Avro und Protocol Buffers; durch die Kompatibilitätsprüfung der Registry erzwungen.
- [ ] **Hat sich die Bedeutung eines Felds geändert?**
  Änderungen von Einheit, Währung oder Semantik brauchen einen neuen Feldnamen, nie einen wiederverwendeten.
- [ ] **Wird der Contract-Schritt nachverfolgt?**
  Legen Sie ein Ticket oder ADR an, damit die alte Struktur tatsächlich entfernt wird, sobald sie ungenutzt ist.

PRÜFEN SIE IHR VERSTÄNDNIS

▼ Ein Team fügt einem Avro-Schema ein Pflichtfeld ohne Standardwert hinzu. Welche Kompatibilität bricht das, und warum?

Die Abwärtskompatibilität: Ein Consumer mit dem neuen Schema, der alte Daten liest, findet das Feld nicht und hat keinen Standardwert zum Einsetzen.

▼ Warum geht der Dual-Write-Schritt weiter, nachdem das Lesen auf die neue Spalte umgestellt wurde?

Damit ein Rollback auf das vorige Release, das noch die alte Spalte liest, diese Spalte auf aktuellem Stand vorfindet.

▼ Warum sollte ein großer Backfill nicht als eine einzige UPDATE-Anweisung in einer Migration geschrieben werden?

Eine riesige Transaktion hält lange Sperren, erzeugt einen Log-Schub, der Replikationsverzögerung verursacht, hinterlässt viel Aufräumarbeit, lässt sich nicht fortsetzen und blockiert das Deployment; kleine, committete Batches vermeiden all das.

▼ Was verhindert es, eine Feldnummer in Protocol Buffers als reserved zu markieren?

Es verhindert, dass jemand die Nummer eines gelöschten Felds für ein neues Feld wiederverwendet, wodurch alte Daten als falsches Feld dekodiert würden.

▼ Probieren Sie es aus: Eine Tabelle bookings hat die Spalte name, und Sie brauchen first_name und last_name. Die Services verwenden Rolling Updates, und ein nächtlicher Job liest name. Listen Sie die Schritte auf, mit denen das ohne Ausfallzeit gelingt, und sagen Sie, warum eine einzelne Migration unsicher ist.

Eine einzige Änderung würde jede laufende Instanz des alten Codes brechen. Verwenden Sie Expand and Contract. Fügen Sie die zwei neuen nullable Spalten hinzu. Deployen Sie Code, der alte und neue schreibt. Füllen Sie alte Zeilen per Backfill in kleinen, gedrosselten, idempotenten Stapeln nach. Stellen Sie die Leser einschließlich des nächtlichen Jobs auf die neuen Spalten um. Hören Sie auf, name zu schreiben, und entfernen Sie die Spalte, sobald nichts sie mehr nutzt. Jeder Schritt funktioniert mit zwei Code-Versionen und kann gefahrlos pausieren.

QUELLEN

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

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

Analytics, Streaming, Data Warehouses, Data Lakes und Data Mesh

BEVOR SIE DAS LESEN

Dieser Abschnitt baut auf:

- 0.5 Daten speichern
- 6.4 Ereignisgesteuert
- 7.1 Datenhoheit

Begriffe von früher, die er verwendet: Datenvertrag (Data Contract), Schema, Fremdschlüssel, Join

DIE IDEE IN KÜRZE

Die Kassen eines Supermarkts sind dafür gebaut, jeden Kunden schnell zu bedienen, einen Einkaufskorb nach dem anderen. Niemand würde das Kassenpersonal bitten, innezuhalten und auszurechnen, welche Produkte sich in den letzten zwei Jahren in jeder Filiale am besten verkauft haben. Stattdessen werden jede Nacht die Tagesumsätze an die Zentrale geschickt, wo Analysten jahrelange Historie bereithalten, aufbereitet für genau diese Art von Frage. Manche Zahlen können nicht bis morgen warten, etwa ein plötzlicher Ansturm auf Brot; sie werden deshalb zusätzlich live beobachtet. Bei der Analytics-Architektur geht es darum, die Zentrale und die Live-Beobachtung aufzubauen, ohne die Kassen zu bremsen.

WAS SIE WISSEN MÜSSEN

- OLTP bedient viele kleine Transaktionen über Schlüssel; OLAP durchsucht und aggregiert große Datenmengen; ihre Trennung schützt die operativen Systeme und erlaubt der Analytics spaltenorientierte, denormalisierte Designs.
- Inmon baut zuerst ein integriertes, normalisiertes unternehmensweites Data Warehouse; Kimball baut dimensionale Sternschemas pro Geschäftsprozess, verbunden durch konforme Dimensionen; jede Faktentabelle braucht eine deklarierte Granularität.
- ELT lädt Rohdaten und transformiert sie im Data Warehouse; Change Data Capture streamt Änderungen aus Datenbank-Logs, aber veröffentlichte Datenverträge sind robuster, als die Tabellen eines anderen Service zu lesen.
- Ein Lakehouse ergänzt offene Dateien im Objektspeicher über Open Table Formats wie Delta Lake, Apache Iceberg und Apache Hudi um Transaktionen, Schemaevolution, Löschungen und Time Travel.

- Stream Processing muss nach Ereigniszeit rechnen, Streams in Tumbling-, Sliding- oder Session-Fenster schneiden und Watermarks sowie eine ausdrückliche Regel für verspätete Daten nutzen.
- Lambda (Marz) kombiniert Batch- und Speed-Layer um den Preis doppelter Logik; Kappa (Kreps, 2014) nutzt einen einzigen Stream-Prozessor und verarbeitet neu, indem es ein aufbewahrtes Log erneut abspielt.
- Data Mesh (Dehghani, 2019) beruht auf fachlicher Datenhoheit, Daten als Produkt, einer Self-Service-Plattform und föderierter, automatisierter Governance; es passt zu großen Organisationen mit reifen Fachteams.

Zwei verschiedene Workloads: OLTP und OLAP

Alles in diesem Kapitel drehte sich bisher um Online-Transaktionsverarbeitung (OLTP): viele kleine, schnelle Lese- und Schreibvorgänge einzelner Datensätze, etwa eine Bestellung aufgeben, eine Zahlung buchen oder die Position eines Kuriers aktualisieren. Analytics stellt eine andere Art von Frage. „Wie hoch war der Umsatz nach Küche und Region, Woche für Woche, in den letzten zwei Jahren?“ berührt Millionen Zeilen, liest von jeder nur wenige Spalten und ändert nichts. Das ist Online Analytical Processing (OLAP), und es unterscheidet sich so stark von OLTP, dass beides auf derselben Datenbank meist keinem von beiden gut dient: Eine große analytische Abfrage kann die Festplatten und den Puffer-Cache auslasten, von denen der Checkout abhängt, und das normalisierte Schema, das Transaktionen korrekt hält, zwingt Analysten, ein Dutzend Tabellen zu joinen.

Die übliche architektonische Antwort besteht darin, die Workloads zu trennen. Operative Systeme behalten ihre eigenen Datenbanken, die ihren Services gehören, wie in Abschnitt 7.1 beschrieben. Daten werden laufend oder periodisch in eine Analyseplattform kopiert, die für Scans ausgelegt ist: spaltenorientierte Speicherung, die nur die benötigten Spalten liest, starke Kompression, massiv parallele Abfrage-Engines und Schemas, die für Analysen geformt sind. Der Rest dieses Abschnitts behandelt, wie diese Plattform entworfen wird, wie Daten zu ihr gelangen, wie Streaming das Bild verändert und wem das Ergebnis gehört.

OLTP und OLAP im Vergleich

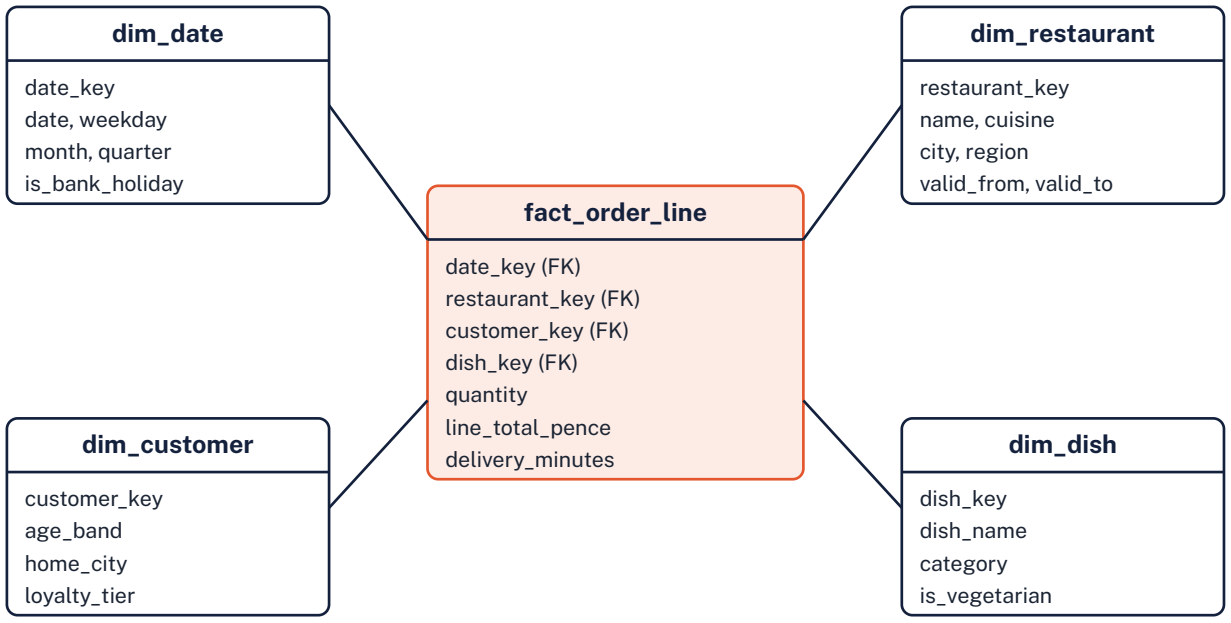
Aspekt	OLTP (operativ)	OLAP (analytisch)
Typische Frage	Bestellung 81723 abrufen; als geliefert markieren	Durchschnittliche Lieferzeit nach Stadt und Stunde, letzte 12 Monate
Berührte Zeilen pro Abfrage	Eine bis einige hundert, über Schlüssel	Millionen bis Milliarden, per Scan
Schreibvorgänge	Ständige kleine Inserts und Updates	Massenladevorgänge oder Anhängen; wenige Updates
Schema	Normalisiert für Integrität (Abschnitt 0.5)	Denormalisiert, dimensional oder breite Tabellen
Speicherlayout	Zeilenorientiert, nach Schlüssel indiziert	Spaltenorientiert, komprimiert, nach Datum partitioniert
Nutzer	Kunden und Anwendungen	Analysten, Data Scientists, Dashboards, ML-Pipelines
Wichtigste Qualitäten	Niedrige Latenz, Konsistenz, Verfügbarkeit	Durchsatz, flexible Abfragen, Historie
Beispiele	PostgreSQL, MySQL, DynamoDB	Snowflake, BigQuery, Redshift, ClickHouse, Databricks SQL

Data Warehouses und dimensionale Modellierung

Ein Data Warehouse ist eine für Analysen gebaute Datenbank, die bereinigte Daten aus vielen operativen Systemen integriert und deren Historie aufbewahrt. Zwei Schulen haben geprägt, wie Data Warehouses entworfen werden, und Fachleute streiten bis heute darüber. **Bill Inmon**, oft der Vater des Data Warehouse genannt (*Building the Data Warehouse*, 1992), vertritt einen Top-down-Entwurf: Zuerst wird ein unternehmensweites, normalisiertes, integriertes Data Warehouse als Single Source of Truth gebaut, und daraus werden dann abteilungsbezogene Data Marts abgeleitet. **Ralph Kimball** (*The Data Warehouse Toolkit*, erstmals 1996 erschienen, dritte Auflage mit Margy Ross 2013) vertritt einen Bottom-up-Entwurf: dimensionale Modelle Geschäftsprozess für Geschäftsprozess bauen und sie über gemeinsame, **konforme Dimensionen** integrieren (eine vereinbarte Definition von Kunde, Datum oder Restaurant, die jedes Modell nutzt). Inmons Ansatz bietet stärkere Integration bei höheren Anfangskosten; Kimballs liefert früher nutzbare Ergebnisse, braucht aber Disziplin, um die Dimensionen konform zu halten. Viele moderne Plattformen verbinden beides: eine bereinigte, integrierte Schicht, die dimensionale Modelle für die Nutzung speist.

Kimballs Sternschema ist nach wie vor das am häufigsten gelehrte Analysemodell. Eine zentrale Faktentabelle erfasst Geschäftsereignisse in einer deklarierten **Granularität** (eine Zeile pro Bestellposition, pro Zahlung, pro Kurier-Ping), mit numerischen Kennzahlen wie Menge, Betrag und Minuten sowie Fremdschlüsseln auf Dimensionstabellen. Jede Dimension enthält beschreibende Attribute zum Filtern und Gruppieren: Datumsangaben (mit vorberechneten Kennzeichen für Wochentag, Monat und Feiertag), Restaurants (Name, Küche, Stadt), Kunden und Gerichte. Abfragen bestehen dann aus einer Faktentabelle, gejoint mit wenigen kleinen Dimensionen, was spaltenorientierte Engines sehr effizient ausführen. Attribute, die sich im Lauf der Zeit ändern, etwa wenn ein Restaurant in eine andere Stadt zieht, werden mit Techniken für **Slowly Changing Dimensions** behandelt; die gängigste, Typ 2, fügt eine neue Dimensionszeile mit Gültigkeitsdaten hinzu, sodass historische Fakten weiterhin auf die Attribute zeigen, die damals galten.

Ein Sternschema für Bestellungen der Essenslieferung



Granularität: eine Zeile pro Gericht je gelieferter Bestellung. Fakten: numerisch, additiv; Dimensionen: beschreibend.

Eine Faktentabelle in einer deklarierten Granularität, umgeben von den Dimensionen, die jeden Fakt beschreiben.

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

Eine typische Sternschema-Abfrage: nach Dimensionsattributen filtern und gruppieren, die Kennzahlen der Fakten aggregieren.

AUSGEARBEITETES BEISPIEL

Ausgearbeitetes Beispiel: die Granularität deklarieren

Das Analytics-Team baut die obige Faktentabelle in der Granularität von einer Zeile pro Gericht einer gelieferten Bestellung. Die Kennzahl `delivery_minutes` auf Bestellebene wurde in jede Position kopiert. Zwei Bestellungen werden geliefert: Bestellung A hat 1 Position und dauerte 20 Minuten; Bestellung B hat 4 Positionen und dauerte 40 Minuten.

Korrekte durchschnittliche Lieferzeit	Pro Bestellung: (20 + 40) ÷ 2 = 30 Minuten.
Durchschnitt über die Faktenzeilen	Positionen: 20, 40, 40, 40, 40. Summe 180 ÷ 5 Zeilen = 36 Minuten. Falsch: Große Bestellungen zählen viermal.
Summe der Lieferminuten	180, was keine Bedeutung hat: Die Kennzahl ist in dieser Granularität nicht additiv.
Lösung	Kennzahlen auf Positionsebene (Menge, Positionssumme) in der Faktentabelle der Bestellpositionen belassen und Kennzahlen auf Bestellebene (Lieferminuten, Liefergebühr, Trinkgeld) in eine separate Tabelle fact_order mit der Granularität von einer Zeile pro Bestellung legen, die dieselben konformen Dimensionen nutzt.
Prüfen	In fact_order: (20 + 40) ÷ 2 = 30 Minuten, und SUM(delivery_fee) zählt jede Gebühr nun einmal.

Kimballs erster Entwurfsschritt besteht darin, die Granularität genau zu deklarieren, bevor Dimensionen oder Fakten gewählt werden, denn jede Kennzahl muss in dieser Granularität stimmen.

Gemischte Granularitäten in einer Tabelle sind eine der häufigsten Ursachen dafür, dass Dashboards einander widersprechen.

Daten hineinbringen: ETL, ELT und Change Data Capture

Daten gelangen über Pipelines ins Data Warehouse. Beim klassischen ETL (Extract, Transform, Load) extrahiert ein separates Werkzeug Daten aus den Quellsystemen, bereinigt und formt sie auf eigenen Servern um und lädt nur das Ergebnis. Seit Cloud-Data-Warehouses Speicher billig und Rechenleistung elastisch gemacht haben, ist **ELT** verbreitet: Rohdaten zuerst extrahieren und laden, dann im Data Warehouse mit SQL transformieren, oft verwaltet als versionierte, getestete Modelle mit einem Werkzeug wie dbt. ELT bewahrt die Rohdaten auf, sodass Transformationen korrigiert und erneut ausgeführt werden können; sein Risiko ist, dass sich personenbezogene Rohdaten weiter verbreiten als nötig.

Wie Daten extrahiert werden, ist ebenso wichtig. Nächtliche Vollkopien sind einfach, aber langsam und veraltet. **Change Data Capture (CDC)** liest mit einem Werkzeug wie Debezium das eigene Replikations-Log der Datenbank (das Write-Ahead-Log von PostgreSQL, das Binlog von MySQL) und streamt jedes Insert, Update und Delete als Event, mit niedriger Latenz und wenig Last auf der Quelle. Unabhängig vom Mechanismus koppelt das direkte Lesen der Tabellen eines anderen Service die Analyseplattform an dessen privates Schema, genau das Problem, das Abschnitt 7.1 beschrieben hat. Das robustere Muster ist, dass jeder Service Daten gezielt veröffentlicht, über Events oder einen vereinbarten Export, im Rahmen eines Datenvertrags.

Data Lakes, Lakehouses und Open Table Formats

Ein Data Lake (der Begriff wird James Dixon zugeschrieben, 2010) speichert Rohdaten aus vielen Quellen in ihrer ursprünglichen Form, meist als Dateien in günstigem Objektspeicher wie Amazon S3, oft in spaltenorientierten Formaten wie Apache Parquet. Er schiebt das Schema auf, bis die Daten gelesen werden; das passt zu Data Scientists, die rohe Details wollen, und zu Daten, deren Verwendung noch nicht bekannt ist. Data Lakes haben sich den Ruf erworben, zu „Datensümpfen“ zu werden: Ohne Verantwortliche, Dokumentation und Qualitätsprüfungen vertraut niemand ihrem Inhalt. Dateien im Objektspeicher fehlt zudem, was ein Data Warehouse bietet: Transaktionen, Updates und Löschungen, Schemadurchsetzung und konsistente Lesevorgänge, während Daten geschrieben werden.

Das Lakehouse (Armbrust, Ghodsi, Xin und Zaharia, CIDR 2021) ergänzt diese Data-Warehouse-Funktionen auf offenen Dateien im Data Lake. Die tragende Technologie ist das Open Table Format: **Delta Lake** (entstanden bei Databricks), **Apache Iceberg** (entstanden bei Netflix) und **Apache Hudi** (entstanden bei Uber). Jedes führt eine Metadatenschicht (ein Transaktions-Log oder einen Baum aus Manifest-Dateien), die festhält, welche Datendateien jede Version einer Tabelle bilden. Das ermöglicht atomare Commits, nebenläufige Schreiber mit optimistischer Nebenläufigkeitskontrolle, Schemaevolution, Updates und Löschungen auf Zeilenebene (nötig für Löschanträge) und **Time Travel**: eine Tabelle so abzufragen, wie sie in einer früheren Version war. Weil das Format offen ist, können mehrere Engines (Spark, Trino, Flink und zunehmend die Cloud-Data-Warehouses selbst) dieselben Tabellen lesen und schreiben, was Lock-in verringert. Viele Lakehouses organisieren Daten in Schichten, von Databricks als Medallion-Architektur bekannt gemacht: Bronze für roh aufgenommene Daten, Silber für bereinigte und konforme Daten, Gold für Modelle auf Geschäftsebene wie Sternschemas.

Stream Processing: Ereigniszeit, Fenster und Watermarks

Batch-Pipelines beantworten Fragen über gestern. Manche Fragen können nicht warten: Wird diese Karte gerade betrügerisch verwendet, sind die Kuriere in Manchester heute Abend verspätet, welche Restaurants sollten pausiert werden, weil ihre Küchen überlastet sind?

Stream-Prozessoren wie Apache Flink, Kafka Streams und Spark Structured Streaming rechnen fortlaufend über unbegrenzte Event-Streams. Die grundlegenden Konzepte stammen aus Googles Dataflow-Modell (Akidau und Kollegen, VLDB 2015).

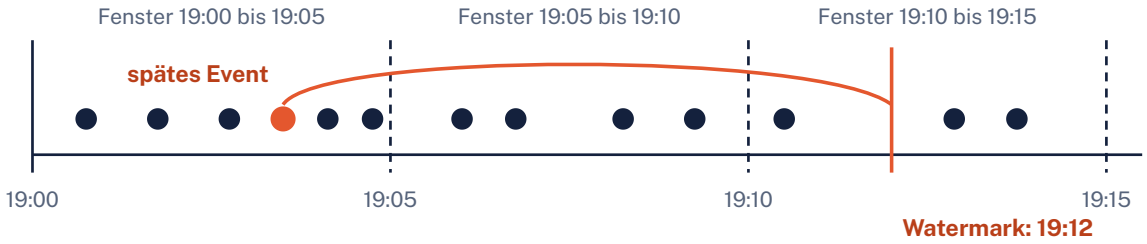
Das erste ist der Unterschied zwischen Ereigniszeit, wann etwas tatsächlich geschah, und **Verarbeitungszeit**, wann das System es beobachtet. Sie weichen voneinander ab wegen Netzwerkverzögerungen, Wiederholungen, Batching und Geräten, die offline gehen: Das Smartphone eines Kuriers im Aufzug erfasst eine Übergabe um 19:03, liefert das Event aber um 19:13 aus. Jedes Ergebnis, das die Wirklichkeit abbilden soll (Bestellungen pro fünf Minuten, durchschnittliche Lieferzeit pro Stunde), muss nach Ereigniszeit berechnet werden, sonst verschiebt es sich, sobald die Pipeline langsam ist, und ist beim erneuten Abspielen falsch.

Das zweite sind **Fenster**, die einen unbegrenzten Stream zum Aggregieren in endliche Stücke schneiden: **Tumbling**-Fenster sind fest und überlappen nicht (alle fünf Minuten), **Sliding**-Fenster überlappen (die letzten zehn Minuten, jede Minute aktualisiert), und **Session**-Fenster gruppieren Aktivitätsschübe pro Schlüssel, getrennt durch eine Lücke der Inaktivität (die Browsing-Sitzung eines Kunden).

Das dritte ist die Watermark: die mitlaufende Schätzung des Prozessors, dass alle Events bis zu einer bestimmten Ereigniszeit wahrscheinlich eingetroffen sind. Wenn die Watermark das Ende eines Fensters überschreitet, kann das Ergebnis des Fensters ausgegeben werden. Watermarks sind Heuristiken, daher treffen manche Events trotzdem ein, nachdem ihr Fenster geschlossen wurde. Die Pipeline braucht eine ausdrückliche Regel für solche **verspäteten Daten**: verwerfen, an eine Nebenausgabe zur späteren Korrektur senden oder Fenster für eine erlaubte Verspätung offen halten und aktualisierte Ergebnisse ausgeben. Längeres Warten liefert vollständigere Antworten, früheres Ausgeben aktuellere. Diese Abwägung zwischen Latenz und Vollständigkeit ist die Stream-Processing-Variante einer Abwägung zwischen Qualitätsmerkmalen und sollte mit dem Fachbereich entschieden werden.

Ereigniszeit, Fenster und ein verspätetes Event

Tumbling-Fenster (5 Min.) nach Ereigniszeit; jeder Punkt steht an seiner Ereigniszeit



Ereigniszeit
wann das Smartphone des Kuriers die Übergabe erfasste: 19:03

Verarbeitungszeit
wann der Stream-Job es sah, nachdem das Handy wieder Netz hatte: 19:13

Das Fenster ab 19:00 war schon zu, als die Watermark 19:05 passierte; das späte Event braucht eine Regel: verwerfen, an eine Nebenausgabe senden oder das Fenster in der erlaubten Verspätung aktualisieren.

Fenster sind über die Ereigniszeit definiert; die Watermark entscheidet, wann sie schließen, und verspätete Events brauchen eine ausdrückliche Regel.

WENN DAS NEU FÜR SIE IST

Schritt für Schritt: zählen nach dem Zeitpunkt des Geschehens

Der schwierigste Gedanke dieses Abschnitts ist, dass ein Stream-Job Events danach gruppieren muss, wann sie geschahen, nicht danach, wann sie eintrafen, und dann entscheiden muss, wie lange er auf Nachzügler wartet. Verfolgen Sie ein verzögertes Event durch eine einfache Zählung.

- Ein Stream-Processing-Job zählt in Fünf-Minuten-Fenstern die Bestellungen, die das Küchentablet jedes Restaurants annimmt, um überlastete Küchen zu erkennen. Zwischen 20:40 und 20:45 nimmt ein gut besuchtes Restaurant 25 Bestellungen an. Sein Tablet verliert kurz nach der Annahme einer Bestellung um 20:43 das WLAN und sendet diese Annahme erst um 20:57.
- Nach Verarbeitungszeit gezählt, also danach, wann der Job jedes Event sieht, zeigt das Fenster ab 20:40 nur 24, und das Fenster ab 20:55 erhält eine Bestellung, die lange vorher angenommen wurde. Bei einem erneuten Abspielen morgen wird der ganze Abend in wenigen Minuten gelesen, und die Zahlen ändern sich wieder.
- Nach Ereigniszeit gezählt, also nach der im Event selbst erfassten Zeit, landet die verzögerte Annahme im Fenster ab 20:40, wo sie hingehört, und ein erneutes Abspielen liefert jedes Mal dasselbe Ergebnis.
- Das wirft eine neue Frage auf: Wann kann der Job die Zahl für 20:40 veröffentlichen, wenn noch ein Nachzügler auftauchen könnte? Er nutzt eine Watermark, seine mitlaufende Schätzung, dass alle Events bis zu einem bestimmten Zeitpunkt wahrscheinlich eingetroffen sind. Wenn die Watermark 20:45 passiert, veröffentlicht der Job 24.
- Um 20:57 trifft die Annahme von 20:43 ein, nachdem ihr Fenster geschlossen wurde. Die Pipeline braucht eine im Voraus festgelegte Regel: verwerfen, für eine spätere Korrektur beiseitelegen oder eine gewisse Verspätung zulassen und eine aktualisierte Zahl von 25 veröffentlichen.
- **Die Lehre:** Rechnen Sie nach Ereigniszeit, damit die Ergebnisse abbilden, was wirklich geschah, und vereinbaren Sie mit dem Fachbereich, wie lange gewartet wird. Längeres Warten liefert vollständigere Zahlen, früheres Veröffentlichen aktuellere.

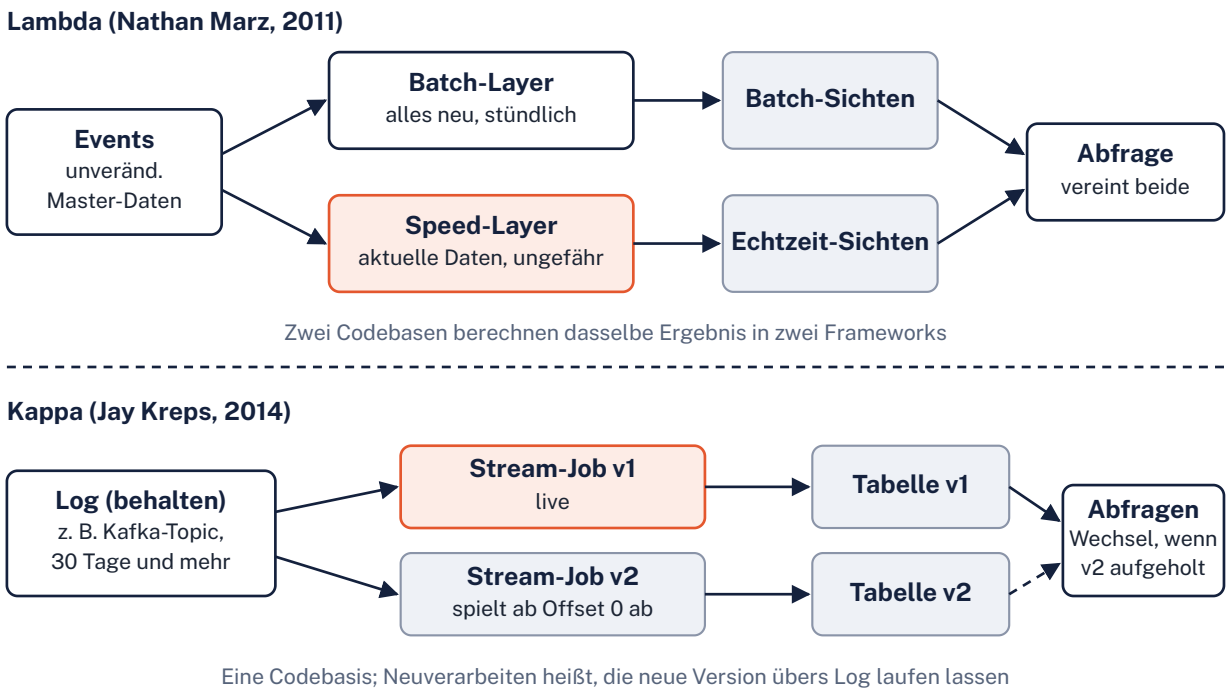
Lambda- und Kappa-Architektur

Wie sollten Batch- und Streaming-Ergebnisse kombiniert werden? Nathan Marz' Essay „How to beat the CAP theorem“ von 2011 beschrieb, was als **Lambda-Architektur** bekannt wurde: Ein unveränderlicher Master-Datenbestand speist einen **Batch-Layer**, der periodisch aus allen Daten genaue Sichten neu berechnet, und einen **Speed-Layer**, der in Echtzeit näherungsweise Sichten auf aktuelle Daten berechnet; Abfragen führen beide zusammen, und jeder Batch-Lauf

ersetzt die Näherungen des Speed-Layers. Lambda bietet Genauigkeit und niedrige Latenz, und der Batch-Layer dient als Sicherheitsnetz für Streaming-Fehler. Der Preis ist, dass dieselbe Logik in zwei verschiedenen Frameworks geschrieben und identisch gehalten werden muss.

2014 stellte Jay Kreps, einer der Schöpfer von Apache Kafka, diesen Entwurf infrage und schlug die **Kappa-Architektur** vor, ein Name, den er halb im Scherz anbot. Nutzen Sie nur einen Stream-Prozessor und bewahren Sie die Eingabe lange genug in einem Log auf, um sie erneut abzuspielen. Um die Logik zu ändern, starten Sie eine zweite Instanz des neuen Jobs, die vom Anfang des Logs in eine neue Ausgabetabelle liest, und stellen die Leser darauf um, sobald sie aufgeholt hat. Eine Codebasis, und Neuverarbeitung ist einfach erneutes Abspielen. Kappa setzt eine lange Log-Aufbewahrung voraus und Stream-Prozessoren, die Ereigniszeit und Zustand zuverlässig korrekt behandeln, was moderne Engines leisten. Heute neigen die meisten neuen Entwürfe zu Streaming im Kappa-Stil mit einem Lakehouse für die Historie, während Entwürfe im Lambda-Stil dort fortbestehen, wo die Batch-Neuberechnung über sehr lange Historien günstiger ist oder wo bestehenden Batch-Pipelines vertraut wird.

Lambda und Kappa nebeneinander



Lambda hält getrennte Batch- und Speed-Pfade und führt sie zur Abfragezeit zusammen; Kappa hat einen einzigen Streaming-Pfad und verarbeitet durch erneutes Abspielen neu.

Data Mesh: Hoheit über analytische Daten

Als Organisationen wuchsen, wurden zentrale Datenteams zu Engpässen: Jede neue Quelle und jede neue Frage stand in der Warteschlange hinter einem Team, das weder die Quellsysteme noch die fachlichen Fragen besonders gut verstand. 2019 schlug Zhamak Dehghani, damals bei

Thoughtworks, das Data Mesh vor (später ausgebaut in ihrem Buch *Data Mesh* von 2022). Es überträgt auf analytische Daten dieselbe Dezentralisierung, die Microservices auf operative Systeme übertragen haben, und beruht auf vier Prinzipien.

Fachliche Datenhoheit: Die Teams, die den Daten am nächsten sind, etwa Bestellungen, Zahlungen oder Kuriere, besitzen und bereitstellen ihre analytischen Daten, so wie sie ihre Services besitzen. **Daten als Produkt**: Jedes Team veröffentlicht Datenprodukte, die auffindbar, dokumentiert, vertrauenswürdig, adressierbar und sicher sind, mit einem Verantwortlichen und Qualitätszielen, und behandelt andere Teams als Kunden. **Self-Service-Datenplattform**: Ein Plattformteam stellt Speicher, Pipelines, Katalog und Zugriffskontrolle bereit, die das Veröffentlichen eines Datenprodukts günstig machen, sodass nicht jede Domäne sie neu erfindet. **Föderierte, automatisierte Governance** (Federated Computational Governance): Globale Regeln (Kennungen, Datenschutzklassifizierungen, Interoperabilitätsstandards) werden gemeinsam vereinbart und von der Plattform automatisch durchgesetzt statt von einem zentralen Prüfgremium.

Data Mesh ist ebenso sehr ein Organisationsdesign wie eine Architektur, und die Meinungen darüber, wann es sich lohnt, gehen auseinander. Seine Befürworter verweisen auf die Skalierung: Es beseitigt den zentralen Engpass und verankert Datenqualität dort, wo das Wissen ist. Seine Kritiker weisen darauf hin, dass es reife Fachteams, eine starke Plattform und echte Investitionen braucht und dass kleinere Organisationen meist mit einer gut geführten zentralen Plattform und klarer Verantwortung für jeden Datenbestand besser bedient sind. Das Gesetz von Conway (Abschnitt 3.7) gilt unmittelbar: Die Datenarchitektur wird die Struktur widerspiegeln, die die Organisation tatsächlich annimmt.

Aufbewahrung, Archivierung und Legal Hold

Daten brauchen neben einem Verantwortlichen auch ein Enddatum. Der Grundsatz der Speicherbegrenzung der UK GDPR verlangt im Allgemeinen, dass personenbezogene Daten in identifizierbarer Form nicht länger als nötig gespeichert werden; jede Datenklasse braucht daher eine mit der Rechtsberatung abgestimmte Aufbewahrungsfrist: Standortverläufe von Kurieren können kurzlebig sein, während Steuerunterlagen Jahre überdauern. **Archivierung** verschiebt Daten, die aufbewahrt werden müssen, aber selten gelesen werden, in günstigeren Speicher und braucht einen Weg, sie zurückzulesen. Die Löschung muss Replikas, Data Warehouses, Event-Logs und Backups erreichen (Abschnitt 8.7). Ein **Legal Hold** setzt den Plan außer Kraft: Unterlagen, die für einen Rechtsstreit relevant sind, müssen möglicherweise aufbewahrt werden, Löschjobs müssen also zuerst nach Holds suchen. Amazon S3 Object Lock zum Beispiel bietet sowohl Aufbewahrungsfristen als auch Legal Holds ohne Ablaufdatum. Modellieren Sie Holds als Daten, nach Kunde oder Bestellung, statt einen Job anzuhalten. Die Fristen hängen vom anwendbaren Recht ab.

Qualität, Lineage, Kataloge und Datenschutz

Analytische Daten sind nur nützlich, wenn man ihnen vertraut. Prüfungen der **Datenqualität** laufen innerhalb der Pipelines und testen Aktualität (ist die heutige Ladung angekommen?), Volumen (hat sich die Zeilenzahl halbiert?), Schema (hat eine Spalte ihren Typ geändert?), Gültigkeit (sind Beträge positiv und Postleitzahlen wohlgeformt?) und Eindeutigkeit. Fehlgeschlagene Prüfungen sollten eine Pipeline anhalten oder Daten in Quarantäne stellen, statt sie unbemerkt zu veröffentlichen. Data Lineage hält fest, woher jeder Datenbestand stammt und welche Transformationen ihn erzeugt haben, sodass Sie, wenn eine Zahl falsch aussieht oder sich ein Quellschema ändert, die Auswirkungen vor- und nachgelagert verfolgen können; die Spezifikation OpenLineage standardisiert, wie Pipelines sie melden. Ein **Datenkatalog** macht Datenbestände auffindbar, mit Beschreibungen, Verantwortlichen, Klassifizierungen und Lineage an einem Ort.

Analyseplattformen bündeln Daten, was sie für Angreifer attraktiv und datenschutzrechtlich riskant macht. Wenden Sie Datenminimierung an (kopieren Sie keine personenbezogenen Felder, die keine Analyse braucht), pseudonymisieren Sie Kennungen, wo möglich, klassifizieren Sie Spalten, damit Zugriffsrichtlinien den Daten folgen, setzen Sie Aufbewahrungsfristen mit automatischer Löschung durch und stellen Sie sicher, dass Löschanträge den Data Lake und das Data Warehouse ebenso erreichen wie die operativen Systeme; Open Table Formats helfen dabei, weil sie Löschungen auf Zeilenebene unterstützen. Abschnitt 8.7 behandelt die Pflichten im Detail.

ACHTUNG

Häufige Fehler

- Schwere analytische Abfragen gegen die produktive OLTP-Datenbank ausführen oder gegen ein Replikat, auf das sich die Anwendung auch für das Failover verlässt.
- Faktentabellen bauen, ohne die Granularität zu deklarieren, sodass Durchschnitte und Summen unbemerkt falsch sind.
- Die privaten Tabellen anderer Services direkt ins Data Warehouse lesen, sodass jede interne Schemaänderung eine Pipeline bricht.
- Streaming-Aggregate nach Verarbeitungszeit berechnen, sodass sich die Ergebnisse ändern, sobald die Pipeline zurückfällt oder erneut abgespielt wird.
- Einen Data Lake als Abladeplatz ohne Verantwortliche, Qualitätsprüfungen oder Katalog behandeln.
- Das Vokabular von Data Mesh übernehmen, ohne die Self-Service-Plattform zu finanzieren oder den Domänen echte Verantwortung zu geben.

PRÜFEN SIE IHR VERSTÄNDNIS

▼ Warum ist es eine schlechte Idee, einen Umsatzbericht über zwei Jahre gegen die Checkout-Datenbank laufen zu lassen?

Der Bericht durchsucht Millionen Zeilen und konkurriert um Festplatte, Arbeitsspeicher und CPU mit den kleinen, latenzempfindlichen Transaktionen, die der Checkout braucht, und das normalisierte Schema macht die Abfrage komplex und langsam.

▼ Welche Entscheidung verlangt Kimballs Methode beim Entwurf einer Faktentabelle als Erstes?

Die Granularität zu deklarieren: genau festzulegen, was eine Zeile darstellt, etwa ein Gericht einer gelieferten Bestellung.

▼ Ein Stream-Job zählt Bestellungen pro Minute nach Verarbeitungszeit. Was passiert, wenn er zehn Minuten zurückfällt und dann aufholt?

Bestellungen werden in den Minuten gezählt, in denen sie verarbeitet wurden, statt in denen sie geschahen; die Zahlen sind also falsch, mit einem Einbruch und danach einer Spitze; Berechnen nach Ereigniszeit vermeidet das.

▼ Wie geht die Kappa-Architektur mit einer Änderung der Verarbeitungslogik um?

Sie lässt die neue Version des Stream-Jobs vom Anfang des aufbewahrten Logs in eine neue Ausgabe laufen, stellt die Leser darauf um, sobald sie aufgeholt hat, und nimmt den alten Job außer Betrieb.

▼ Probieren Sie es aus: Eine Faktentabelle hat eine Zeile pro Ticket, und jede Zeile kopiert die Gebühr der Buchung. Buchung A hat 1 Ticket und eine Gebühr von £3; Buchung B hat 3 Tickets und eine Gebühr von £5. Vergleichen Sie die korrekten und die naiven Ergebnisse. Was würden Sie ändern?

Richtig: Gebühren gesamt £8, Durchschnitt £4 pro Buchung. Summe der Faktenzeilen: 3 + 5 + 5 + 5 = £18; ihr Durchschnitt: 18 / 4 = £4,50. Die Gebühr gehört auf die Granularität der Buchung, nicht des Tickets. Legen Sie die Granularität fest, und führen Sie dann entweder eine eigene Faktentabelle mit einer Zeile pro Buchung oder verteilen Sie die Gebühr auf die Tickets, sodass aus £5 etwa je £1,67 werden, und kennzeichnen Sie das klar.

QUELLEN

- Zhamak Dehghani: How to Move Beyond a Monolithic Data Lake to a Distributed Data Mesh (2019): https://martinfowler.com/articles/data-monolith-to-mesh.html

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

Weiterführende Literatur

- *Designing Data-Intensive Applications (2nd edition)*, Martin Kleppmann and Chris Riccomini (2026), O'Reilly Media. Das maßgebliche moderne Standardwerk zu Replikation, Partitionierung, Transaktionen, Stream Processing und abgeleiteten Daten; die meisten Mechanismen dieses Kapitels werden dort ausführlich erklärt.
  https://martin.kleppmann.com/2026/03/24/designing-data-intensive-applications-2e.html
- *Software Architecture: The Hard Parts*, Neal Ford, Mark Richards, Pramod Sadalage and Zhamak Dehghani (2021), O'Reilly Media. Datenhoheit, Techniken für gemeinsame Datenhoheit, Muster für verteilten Datenzugriff sowie Kräfte, die Daten auftrennen oder zusammenführen, mit einer durchgehenden Fallstudie.
- *A Critique of ANSI SQL Isolation Levels*, Hal Berenson, Phil Bernstein, Jim Gray, Jim Melton, Elizabeth O'Neil and Patrick O'Neil (1995), Microsoft Research technical report MSR-TR-95-51 (also ACM SIGMOD 1995). Das kurze Paper, das Snapshot Isolation und die im SQL-Standard fehlenden Anomalien definierte.
  https://www.microsoft.com/en-us/research/publication/a-critique-of-ansi-sql-isolation-levels/
- *Dynamo: Amazon's Highly Available Key-value Store*, Giuseppe DeCandia and colleagues (2007), ACM Symposium on Operating Systems Principles (SOSP). Das Paper hinter Leaderless Replication, Quoren und konsistentem Hashing mit virtuellen Knoten in Cassandra und verwandten Systemen.
  https://www.allthingsdistributed.com/files/amazon-dynamo-sosp2007.pdf
- *Choose Boring Technology*, Dan McKinley (2015), mcfunley.com. Ein kurzer, einflussreicher Essay über Innovation Tokens und die langfristigen Kosten zusätzlicher Technologien, besonders von Datenbanken.
  https://mcfunley.com/choose-boring-technology
- *The Data Warehouse Toolkit (3rd edition)*, Ralph Kimball and Margy Ross (2013), Wiley. Das maßgebliche Handbuch zur dimensionalen Modellierung, zu Sternschemata sowie Fakten- und Dimensionstabellen für Analytics.
- *Data Mesh: Delivering Data-Driven Value at Scale*, Zhamak Dehghani (2022), O'Reilly Media. Die buchlange Darstellung domänenverantworteter analytischer Daten, von Daten als Produkt, einer Self-Service-Plattform und föderierter Governance.

Glossar

Abwärtskompatibilität

Eine Schemaänderung ist abwärtskompatibel, wenn Code, der das neue Schema verwendet, weiterhin Daten lesen kann, die mit dem alten geschrieben wurden.

Anomalie (Phänomen)
Ein falsches Ergebnis, weil sich nebenläufige Transaktionen gegenseitig stören, etwa ein Dirty Read, ein Lost Update oder ein Write Skew; Isolationsebenen werden dadurch definiert, welche Anomalien sie verhindern.

Aufwärtskompatibilität
Eine Schemaänderung ist aufwärtskompatibel, wenn Code, der das alte Schema verwendet, weiterhin Daten lesen kann, die mit dem neuen geschrieben wurden, meist indem er unbekannte Felder ignoriert.

Backfill
Das Befüllen einer neuen Spalte oder Tabelle mit Werten für bestehende Zeilen, meist in kleinen Stapeln, damit das Live-System nicht gestört wird.

Cache Stampede
Ein Ansturm von Anfragen, die alle im selben Moment den Cache verfehlen, meist wenn ein beliebter Eintrag abläuft, und alle gleichzeitig die langsamere Quelle treffen; eng verwandt mit dem Thundering-Herd-Problem.

Cache-Aside
Ein Caching-Muster, bei dem die Anwendung den Cache prüft, bei einem Fehltreffer aus der Datenbank lädt und das Ergebnis selbst in den Cache legt; auch Lazy Loading genannt.

Cache-Invalidierung
Das Entfernen einer zwischengespeicherten Kopie oder ihre Kennzeichnung als veraltet, wenn sich die Originaldaten ändern, damit spätere Lesezugriffe den neuen Wert holen.

Cache-Trefferquote
Der Anteil der Abfragen, die der Cache aus seinem eigenen Inhalt beantworten kann; der Rest sind Fehltreffer (Misses), die an die langsamere Quelle gehen.

Command-Query Separation (CQS)
Das Prinzip von Bertrand Meyer, nach dem eine Methode entweder den Zustand ändern (ein Command) oder eine Antwort zurückgeben (eine Query) soll, aber nicht beides.

Crypto-Shredding
Daten unlesbar machen, indem der Schlüssel vernichtet wird, mit dem sie verschlüsselt sind; verwendet, um personenbezogene Daten aus Logs und Backups zu löschen, die sich nicht bearbeiten lassen.

Data Lake
Ein großer, günstiger Speicher, meist Objektspeicher, der Rohdaten aus vielen Quellen in ihrer ursprünglichen Form für spätere Analysen aufbewahrt.

Data Lineage
Eine Aufzeichnung darüber, woher ein Datum stammt und welche Transformationen es auf dem Weg durchlaufen hat.

Data Mesh
Der Ansatz von Zhamak Dehghani für analytische Daten, bei dem Domänenteams ihre Daten als Produkte auf einer Self-Service-Plattform unter föderierter Governance verantworten und veröffentlichen.

Data Warehouse
Eine für Analysen gebaute Datenbank, die bereinigte Daten aus vielen operativen Systemen zusammenführt und ihre Historie aufbewahrt.

Datenprodukt
Ein von einem Domänenteam zur Nutzung durch andere veröffentlichter Datensatz mit Verantwortlichem, Dokumentation, Qualitätsgarantien und einer stabilen Schnittstelle.

Datenvertrag (Data Contract)
Eine ausdrückliche, versionierte Vereinbarung zwischen einem Dateneigentümer und seinen Consumern über Schema, Bedeutung, Qualität, Aktualität, Zuständigkeit und die Ankündigung von Änderungen.

Dimensionstabelle
Eine Tabelle beschreibender Attribute, etwa Name, Küche und Stadt eines Restaurants, die zum Filtern und Gruppieren von Fakten dient.

Distributed SQL (NewSQL)
Relationale Datenbanken wie Spanner und CockroachDB, die SQL und ACID-Transaktionen beibehalten und Daten zugleich über viele Knoten oder Regionen partitionieren und replizieren.

Ereigniszeit (Event Time)
Der Zeitpunkt, zu dem ein Ereignis tatsächlich stattfand, im Gegensatz zur Verarbeitungszeit, zu der ein System es wahrnimmt.

ETag
Ein HTTP-Antwortheader mit einer Kennung für eine bestimmte Version einer Ressource, die ein Client in If-None-Match zurücksenden kann, um zu fragen, ob sie sich geändert hat.

Event-Stream
Die geordnete, nach Version nummerierte Folge von Ereignissen eines Aggregats, aus der der Zustand dieses Aggregats wiederhergestellt wird.

Expand and Contract
Einen Breaking Change in sicheren Schritten durchführen: die neue Struktur neben der alten hinzufügen, Leser, Schreiber und Daten migrieren und dann die alte Struktur entfernen. Auch Parallel Change genannt.

Faktentabelle
Die zentrale Tabelle eines Sternschemas mit einer Zeile pro Geschäftsereignis in einer festgelegten Granularität, mit numerischen Kennzahlen und Schlüsseln zu den Dimensionen.

Gemeinsame Datenhoheit (Joint Ownership)
Eine Situation, in der mehrere Services dieselben Daten schreiben. Sie lässt sich auflösen, indem man die Tabelle aufteilt, eine kleine gemeinsame Datendomäne bildet, alle Schreibzugriffe an einen Eigentümer delegiert oder die Services zusammenlegt.

Integrationsdatenbank
Eine von mehreren Anwendungen gemeinsam genutzte Datenbank, über die sie sich integrieren, indem sie dieselben Tabellen lesen und schreiben. Martin Fowler (2004) rät davon ab, weil sie die Anwendungen eng koppelt.

Isolationsebene
Eine Einstellung, die angibt, wie stark sich nebenläufige Transaktionen gegenseitig beeinflussen dürfen: Read Uncommitted, Read Committed, Repeatable Read, Snapshot oder Serializable. Die Namen bedeuten in verschiedenen Datenbanken Unterschiedliches.

Konsistentes Hashing
Ein Verfahren, Schlüssel auf einem Hash-Ring Knoten zuzuordnen, sodass beim Hinzufügen oder Entfernen eines Knotens nur etwa 1/N der Schlüssel verschoben werden (Karger und Kollegen, 1997).

Lakehouse
Eine Architektur, die Data-Warehouse-Funktionen wie Transaktionen, Schemas und schnelle Abfragen für Daten bereitstellt, die in offenen Dateiformaten auf Object Storage liegen.

Leader-Follower-Replikation
Ein Replikationsverfahren, bei dem ein Knoten (der Leader oder Primary) alle Schreibvorgänge annimmt und seine Änderungen an Follower (Replikate) streamt, die Lesezugriffe bedienen können.

Leaderless Replication
Ein Verfahren im Stil von Dynamo, bei dem Clients direkt auf mehrere Replikate schreiben und von mehreren lesen und statt eines einzelnen Leaders Quoren verwenden.

Least Frequently Used (LFU)
Eine Verdrängungsstrategie, die bei vollem Cache das Element entfernt, das am seltensten gelesen wurde.

Least Recently Used (LRU)
Eine Verdrängungsstrategie, die bei vollem Cache das Element entfernt, das am längsten nicht mehr gelesen wurde.

Lost Update
Eine Anomalie, bei der zwei Transaktionen einen Wert lesen, jede einen neuen berechnet und zurückschreibt, sodass eine Änderung die andere stillschweigend überschreibt.

Migration
Ein versioniertes Skript, das ein Datenbankschema oder dessen Daten von einer Version auf die nächste bringt, in der Versionskontrolle liegt und automatisch angewendet wird.

Multi-Leader-Replikation
Ein Verfahren, bei dem mehrere Knoten Schreibvorgänge annehmen und untereinander replizieren, was eine Möglichkeit erfordert, widersprüchliche Schreibvorgänge zu erkennen und aufzulösen.

Multi-Version Concurrency Control (MVCC)
Eine Technik, bei der Schreibvorgänge neue Zeilenversionen anlegen, statt zu überschreiben, sodass Leser einen konsistenten Snapshot sehen, ohne Schreiber zu blockieren.

OLAP
Online Analytical Processing: Workloads aus großen Abfragen, die viele Datensätze durchsuchen und aggregieren, etwa Umsatz nach Region und Monat.

OLTP
Online Transaction Processing: Workloads aus vielen kleinen Lese- und Schreibzugriffen auf einzelne Datensätze, etwa das Aufgeben von Bestellungen oder das Verbuchen von Zahlungen.

Open Table Format
Eine Spezifikation wie Apache Iceberg, Delta Lake oder Apache Hudi, die Datendateien und Metadaten so organisiert, dass viele Engines sie als Tabellen mit Transaktionen und Time Travel behandeln können.

Optimistische Nebenläufigkeitskontrolle
Konflikte erkennen statt verhindern: Ein Schreibvorgang gelingt nur, wenn eine Versionsnummer oder ein Zeitstempel seit dem Lesen der Daten unverändert ist; andernfalls versucht der Schreibende es erneut.

Partitionierung (Sharding)
Die Aufteilung eines Datenbestands, sodass jede Maschine nur einen Teil davon hält, ausgewählt über einen Partitionsschlüssel, um Schreibvorgänge und Speicher über eine Maschine hinaus zu skalieren.

Pessimistisches Sperren
Konflikte verhindern, indem Zeilen beim Lesen gesperrt werden (zum Beispiel mit SELECT … FOR UPDATE), sodass andere Transaktionen warten, bis die Sperre aufgehoben ist.

Polyglotte Persistenz
Der Einsatz verschiedener Datenbankarten für verschiedene Daten innerhalb eines Systems, ausgewählt danach, wie jeder Datenbestand genutzt wird, um den Preis, mehr Technologien betreiben zu müssen.

Projektion
Code, der auf Änderungen oder Ereignisse lauscht und ein Lesemodell aktuell hält; auch Bezeichnung für das so erzeugte Lesemodell.

Replikation
Das Vorhalten von Kopien derselben Daten auf mehreren Maschinen für Verfügbarkeit, Lesekapazität und geringere Latenz in Nutzernähe.

Replikationsverzögerung (Replication Lag)
Die Verzögerung zwischen einem Schreibvorgang auf dem Leader und seinem Erscheinen auf einem Follower; sie führt dazu, dass Nutzer veraltete Daten lesen.

Request Coalescing
Nur eine Anfrage holt ein fehlendes Element, während andere Anfragen nach demselben Element auf ihr Ergebnis warten und es mitnutzen; auch Single Flight genannt.

Schema Registry
Ein Dienst, der jede Version jedes Nachrichtenschemas speichert und neue Versionen ablehnt, die gegen die vereinbarte Kompatibilitätsregel verstoßen.

Sekundärindex
Ein Index auf ein anderes Attribut als den Primär- oder Partitionsschlüssel. In einem partitionierten Speicher ist er entweder lokal für jede Partition oder global und nach dem indexierten Wert partitioniert.

Serialisierbarkeit
Die stärkste Isolationsgarantie: Nebenläufige Transaktionen liefern dasselbe Ergebnis, als wären sie in irgendeiner Reihenfolge nacheinander ausgeführt worden.

Snapshot
Eine gespeicherte Kopie des Zustands eines Aggregats bei einer bekannten Event-Version, die verhindert, dass ein langer Stream von Anfang an neu abgespielt werden muss; sie ist ein Cache, keine maßgebliche Datenquelle.

Snapshot Isolation
Eine Isolationsebene, bei der jede Transaktion ab ihrem Beginn einen konsistenten Snapshot der bestätigten Daten liest und widersprüchliche Schreibvorgänge auf dieselbe Zeile abgewiesen werden. Sie verhindert Lost Updates, lässt aber Write Skew zu.

Split Brain
Ein Fehlerzustand, bei dem zwei Knoten beide glauben, der Leader zu sein, und beide Schreibvorgänge annehmen, sodass ihre Daten auseinanderlaufen.

Sternschema
Ein dimensionales Modell mit einer zentralen Faktentabelle aus Messwerten, umgeben von Dimensionstabellen, die diese beschreiben, etwa Datum, Kunde und Produkt.

Suchmaschine
Ein Speicher, der auf einem invertierten Index von Wörtern auf Dokumente aufbaut und eine nach Relevanz sortierte, tippfehlertolerante Volltextsuche bietet; meist aus einem System of Record gespeist.

System of Record
Die maßgebliche Quelle für ein Datum, deren Wert gilt, wenn Kopien voneinander abweichen, etwa das Kernbuchungssystem einer Bank für Kontostände.

Time to Live (TTL)
Wie lange ein zwischengespeichertes Element verwendet werden darf, bevor es als veraltet gilt und aufgefrischt oder neu validiert werden muss.

Tolerant Reader
Ein Consumer, der nur die Felder liest, die er braucht, und alles ignoriert, was er nicht kennt, sodass Anbieter Felder hinzufügen können, ohne ihn zu beschädigen.

Upcasting
Die Umwandlung eines in einer alten Schemaversion gespeicherten Ereignisses in die aktuelle Version beim Lesen, sodass der Anwendungscode nur eine Form verarbeiten muss.

Vektordatenbank
Ein Speicher für Embeddings (Zahlenlisten, die Bedeutung repräsentieren), der die einer Abfrage ähnlichsten Elemente findet, meist mit Indizes für die approximative Nächste-Nachbarn-Suche.

Watermark
Die fortlaufende Schätzung eines Stream-Prozessors, dass alle Ereignisse bis zu einer bestimmten Ereigniszeit wahrscheinlich eingetroffen sind, anhand derer entschieden wird, wann das Ergebnis eines Fensters ausgegeben werden kann.

Write Skew
Eine Anomalie, bei der zwei Transaktionen sich überschneidende Daten lesen, jede eine Änderung für sicher hält und sie verschiedene Zeilen aktualisieren, wodurch sie gemeinsam eine Regel verletzen, die jede einzeln geprüft hat.

Write-Ahead-Log (WAL)
Ein Log auf der Festplatte, an das nur angehängt wird und in dem eine Datenbank jede Änderung vor ihrer Anwendung festhält, sodass bestätigte Arbeit einen Absturz übersteht; Replikate werden oft daraus gespeist.

Write-behind-Cache
Ein Caching-Muster, bei dem Schreibvorgänge bestätigt werden, sobald sie den Cache erreichen, und erst später im Hintergrund in den darunterliegenden Speicher geschrieben werden; auch Write-back genannt.

Write-through-Cache
Ein Caching-Muster, bei dem jeder Schreibvorgang in den Cache und in den darunterliegenden Speicher gelangt, bevor er bestätigt wird, sodass der Cache nie älter ist als der Speicher.

Zeitreihendatenbank
Eine Datenbank, optimiert für zeitgestempelte Messwerte, die der Reihe nach eintreffen, mit Komprimierung, Downsampling und kostengünstigem Löschen alter Daten.