Data Engineering

NoSQL

NOSQL

- Steht für “Not only SQL“
- Nicht relationale Datenbanken mit anderen Datenstrukturen
- Warum?

Wichtigkeit von Datenmodellen:

- Zentrale Rolle in der Softwareentwicklung
- Beeinflussen, wie Software geschrieben wird
- Prägen, wie Probleme verstanden und gelöst werden

Schichten von Datenmodellen:

- Anwendung baut auf mehreren Datenmodell-Schichten auf
- **Schlüsselfrage**: Wie wird ein Modell auf der nächsttieferen Schicht repräsentiert?

Schichtenmodell Beispiele:

1. **Anwendungsebene**:
    1. Entwickeln von Modellen basierend auf der realen Welt (Menschen, Organisationen, Aktionen, etc.)
    2. Modellierung erfolgt in Form von Objekten, Datenstrukturen und APIs
    3. Strukturen sind oft spezifisch für die Anwendung
2. **Datenspeicherungsebene**:
    1. Datenstrukturen werden in ein allgemeines Datenmodell überführt (z.B. JSON, XML, relationale Tabellen, Graphen)
3. **Physische Ebene**:
    1. Datenbank-Ingenieure definieren, wie diese Modelle in Bytes (im Speicher, auf Festplatten oder im Netzwerk) repräsentiert werden
    2. Repräsentation ermöglicht Abfragen, Suche, Manipulation und Verarbeitung

Arten von NOSQL Datenbanken

- Key-value Stores (Redis)
- Wide-column Stores (Cassandra)
- Graphen Datenbanken (Neo4J)
- Dokumentenorientierte Datenbanken (MongoDB, Elasticsearch)
- Vectordatenbanken (pgvector, elasticsearch, mongodb)
- uvm.
- **Heute:** Viele Datenbanken sind multimodal

Historie

Relationales Modell

- Basierend auf dem **relationalen Modell** von Edgar Codd (1970)
- **Datenorganisation**: Relationen (Tabellen) mit einer Sammlung von Zeilen
- Anfangs Zweifel an effizienter Implementierung, aber ab Mitte der 1980er **dominierende Technologie**

Vorteile

- Breite Anwendbarkeit über den ursprünglichen Fokus der **geschäftlichen Datenverarbeitung** hinaus
- **Dominanz im Web**: Angetrieben von relationalen Datenbanken für Online-Publishing, Social Networking, E-Commerce, SaaS und mehr
- Das beste Modell für eine große Anzahl an verschiedenen Usecases

Entstehung von NoSQL

- Aufgekommen in den 2010er Jahren als Reaktion auf die **Dominanz des relationalen Modells**
- **Bezeichnung "NoSQL"**: Ursprünglich ein Twitter-Hashtag für ein Treffen zu Open-Source, verteilten, nicht-relationalen Datenbanken (2009)
- Später als **"Not Only SQL"** interpretiert

Treibende Kräfte hinter NoSQL

1. **Skalierbarkeit**: Bedarf an höherer Skalierbarkeit für große Datensätze und hohen Schreibdurchsatz
2. **Open-Source Präferenz**: Bevorzugung von kostenfreien Open-Source-Datenbanken gegenüber kommerziellen Produkten
3. **Spezialisierte Abfragen**: Relationale Modelle unterstützen bestimmte Abfragen schlecht
4. **Dynamische Datenschemata**: Wunsch nach flexibleren und ausdrucksstärkeren Datenmodellen als in relationalen Datenbanken

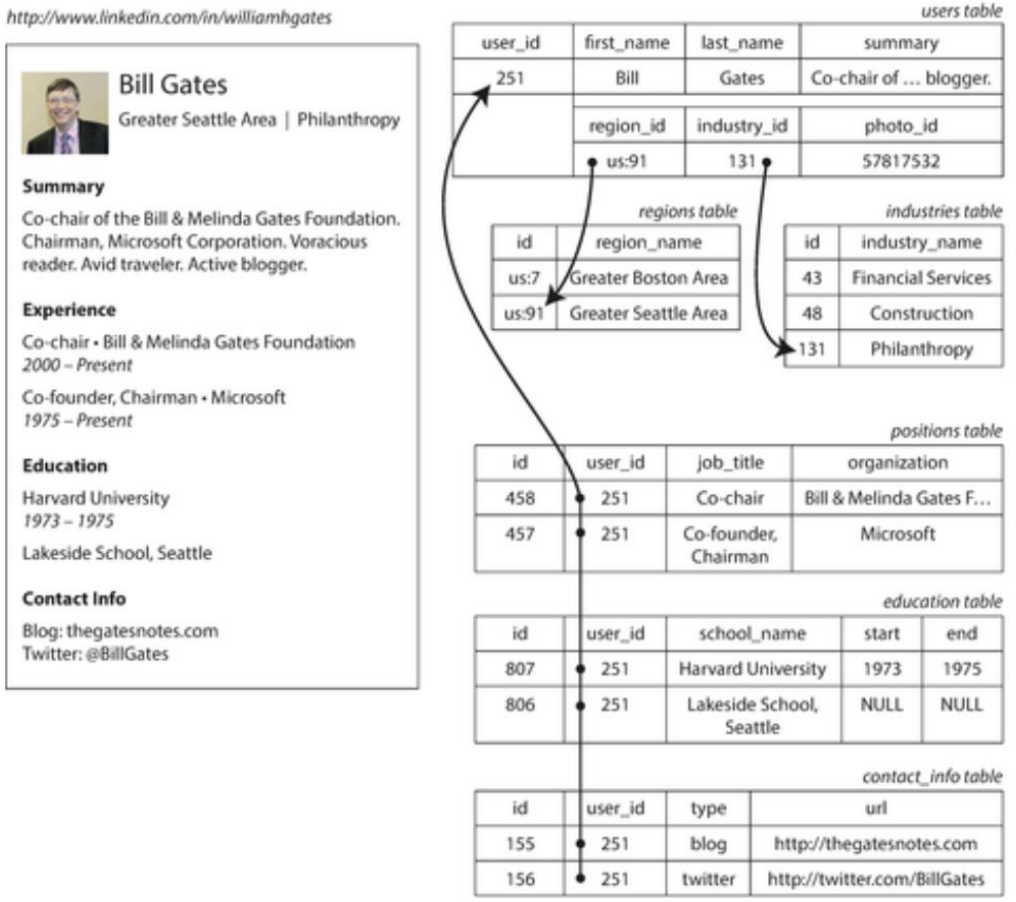
Warum NoSQL

Am Beispiel Dokumentbasierte Datenbanken

Kritik am SQL-Datenmodell

- Da viele Anwendungen heute in **objektorientierten Programmiersprachen** entwickelt werden, entsteht ein Problem bei der **Übersetzung zwischen Objekten** im Code und den relationalen Modellen (Tabellen, Zeilen, Spalten) der Datenbank.
- Diese **Diskrepanz** wird als **"Impedance Mismatch"** bezeichnet und stellt einen erheblichen Übersetzungsaufwand dar.
- object-relational mapping Frameworks wie Hibernate lösen dieses Problem bedingt

Beispiel einer relationalen Darstellung eines Lebenslaufs (LinkedIn-Profil):



Dokumentenorientierte Datenbanken

Beispiel einer relationalen Darstellung eines Lebenslaufs (LinkedIn-Profil):

- Ein **Lebenslauf** wird in relationalen Datenbanken durch einen **eindeutigen Identifikator (user_id)** repräsentiert.
- **Felder**, die pro Benutzer nur einmal vorkommen (z.B. **Vorname, Nachname**), werden als Spalten in der **Benutzer-Tabelle** gespeichert.
- Für Informationen wie **Jobs, Bildungseinträge** oder **Kontaktinformationen**, die in unterschiedlichen Mengen vorkommen können, werden **One-to-Many-Beziehungen** genutzt und diese Daten in separaten Tabellen gespeichert.
- Für solche komplexen, **selbstenthaltenden Datenstrukturen** wie einen Lebenslauf ist eine **JSON-Repräsentation** oft eine geeignetere Lösung, da sie einfacher zu verwalten ist.

Dokumentenorientierte Datenbanken

- Datenbanken wie **MongoDB, RethinkDB, CouchDB** und **Espresso** unterstützen das **dokumentenorientierte Modell** und sind auf die Speicherung von **JSON-Datenstrukturen** ausgelegt.
- Diese **NoSQL-Datenbanken** bieten mehr Flexibilität bei der Speicherung und Verarbeitung solcher unstrukturierter oder semi-strukturierter Daten als relationale Systeme.

Dokumentenorientierte Datenbanken:

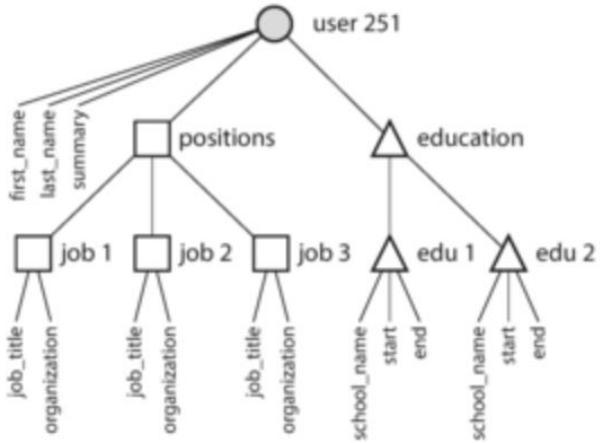
```
{
  "user_id":     251,
  "first_name":  "Bill",
  "last_name":   "Gates",
  "summary":     "Co-chair of the Bill & Melinda Gates... Active blogger.",
  "region_id":   "us:91",
  "industry_id": 131,
  "photo_url":   "/p/7/000/253/05b/308dd6e.jpg",
  "positions": [
    {"job_title": "Co-chair", "organization": "Bill & Melinda Gates Foundation"},
    {"job_title": "Co-founder, Chairman", "organization": "Microsoft"}
  ],
  "education": [
    {"school_name": "Harvard University",       "start": 1973, "end": 1975},
    {"school_name": "Lakeside School, Seattle", "start": null, "end": null}
  ],
  "contact_info": {
    "blog":    "http://thegatesnotes.com",
    "twitter": "http://twitter.com/BillGates"
  }
}
```

JSON vs. Relational Modeling

- Viele Entwickler bevorzugen das **JSON-Modell**, weil es den **Disconnect** („impedance mismatch“) zwischen Applikationscode und Speicherstruktur reduziert. JSON hat jedoch auch **Nachteile**, wie das Fehlen eines Schemas
- Ein wesentlicher **Vorteil** von JSON ist die **Datenlokalität** ;Alle Profilinformationen werden an einem Ort gespeichert, während im relationalen Modell mehrere **Abfragen** oder **Joins** erforderlich sind, um Daten aus verschiedenen Tabellen abzurufen
- Die Daten liegen vollständig denormalisiert dar

Baumstruktur und One-To-Many

- Die one-to-many Beziehungen zwischen Benutzerprofilen, Stellenangeboten, Ausbildungsverlauf und Kontaktinformationen bilden eine Baumstruktur in den Daten. JSON macht diese Struktur explizit und vereinfacht die Modellierung im Vergleich zu relationalen Modellen



Many-to-Many

- Von Many-to-Many Beziehungen spricht man dann wenn mehrere Instanzen einer Entität mit mehreren Instanzen einer anderen Entität in Beziehung steht
- Ein Beispiel sind Unternehmen und Nutzer bei Linkedin
  - => Nutzer haben mehrere Arbeitgeber, ein Unternehmen beschäftigt gleichzeitig mehrere Nutzer
- Dies ist in unserer skizzierten Baumstruktur nicht abbildbar

Many-to-Many

- Viele Dokumentendatenbanken (z. B. Elasticsearch) unterstützen keine Joins!
- Obwohl denormalisierte Datenstrukturen anfangs gut funktionieren, werden die Daten im Laufe der Zeit immer vernetzter →Führt in der Praxis teilweise zu unlösbaren Modelierungsproblemen

Weitere Nachteile

- Denormaliserte Speicherung von Daten führt tendenziell zu:
  - Inkonsistenzen
  - Höherem Speicherbedarf
  - **Sehr teuren Updates**
  - Teureren Writes
- Teure Updates Aufgrund des Speichermanagements (LSM Trees)
- Keine ACID Transaktionen
- Keine einheitliche API

Schema-on-Read and Schema Changes

- Schema-on-Read bietet Flexibilität beim Speichern neuer Felder, ohne dass eine vollständige Migration erforderlich ist, was bei dynamischen oder heterogenen Datenmodellen von Vorteil ist.
- In relationalen Datenbanken können Schema-Änderungen über Befehle wie „ALTER TABLE“ vorgenommen werden, die jedoch langsam sein und bei großen Datenmengen Ausfallzeiten verursachen können.

Data Locality

- Die Speicherung eines Dokuments als fortlaufenden String bietet Vorteile, wenn häufig auf das gesamte Dokument zugegriffen wird
- Bei relationalen Datenbanken, müssen durch die Joins mehrere Tabellen gelesen werden
- Updates sind hingegen extrem kostspielig, da hierfür das gesamte Dokument neu geschrieben werden muss

Wichtige Konzepte

Skalierbarkeit

Skalierbarkeit

- Vertikale Skalierung: scale up
- Steigerung der Leistung durch hinzufügen weiterer Ressourcen zu **einer Instanz** (CPU, RAM, etc.)
- **Con:** Nur bedingt möglich und ab einem bestimmten Punkt unwirtschaftlich
- **Pro:** Sehr einfach ohne einschneidende Änderungen möglich

- Horizontale Skalierung: Scale Out
- Skalierung durch hinzufügen weiterer **Instanzen**
- **Pro:** theoretisch unendliche Skalierbarkeit, automatisierbar
- **Con:** Nicht jede Operation oder Datenstruktur lässt sich parallelisieren. Scale Out erfordert eine skalierbare Architektur

Datenbank-Sharding

- Daten werden über mehrere Datenbank Instanzen (unabhängige „Computer“) gespeichert
- Die Datenbank stellt dabei weiterhin ein konsolidiertes System dar
- Grund: Eine Instanz einer Datenbank kann nur eine begrenzte Anzahl an Daten halten und verarbeiten
- *Availability:* Datenbanken mit mehreren Instanzen können im Falle von Systemausfällen einzelner Instanzen weiter funktionieren

Shards – Beispiel relational

- Datenbanken halten Informationen in Tabellen, die wiederum aus Zeilen und Spalten bestehen
- Idee: Inhalt der Tabellen in *Chunks* schneiden und über mehrere Instanzen verteilen => Shards oder auch Partitionen
- Alle Operationen werden dann auf den einzelnen Datenbankinstanzen ausgeführt → Nur das Schema wird geteilt

Shards – Beispiel relational

- Naiver Ansatz: Wir teilen die Tabelle bei der Hälfte
- Jeder Node (Instanz) erhält die hälfte der Daten

Kunden-ID	Name	Bundesstaat
1	John	Kalifornien
2	Jane	Washington
3	Paulo	Arizona
4	Wang	Georgia

Computer A

Kunden-ID	Name	Bundesstaat
1	John	Kalifornien
2	Jane	Washington

Computer B

Kunden-ID	Name	Bundesstaat
3	Paulo	Arizona
4	Wang	Georgia

Herausforderung Sharding

- **Abhängigkeiten:** Müssen zur Beantwortung von Anfragen Daten mehrerer Knoten verarbeitet werden, leidet die Performance drastisch
  - Daten müssen über das Netzwerk ausgetauscht werden => langsam
  - Knoten sind auf die Abarbeitung von anderen Knoten abhängig => Sequentielle Abarbeitung statt paralleler Abarbeitung
- **Daten Lokalität:** Beim Schneiden der Shards ist es extrem wichtig zu bedenken welche Daten sich auf welchem Node befinden
  - Zu beachten ist: Welche Anfragen werden an das System gestellt? Welche Daten werden oft zusammen abgefragt?

Shard Schlüssel

- Um Daten clever über das System zu verteilen werden Schlüssel verwendet
- Mit Hilfe des Schlüssels wird entschieden welche Daten in unmittelbarer Nähe zueinander gespeichert werden
- Dafür wird (je nach DB) häufig der Wert einer Spalte herangezogen
- Im gezeigten Beispiel wäre (je nach Usecase) der Bundesstaat ein probates Mittel
- **Charmant:** Die Anwendung selbst kann aufgrund des Schlüssels entscheiden welchen Datenbank-Node sie anspricht

Shard Schlüssel – Herausforderung

- Damit gewählte Schlüssel keine Probleme verursachen müssen gewisse Bedingungen erfüllt sein
- Der Schlüssel muss dem Usecase entsprechend gewählt werden
  - Beispiel: Todolisten App: Operationen ausschließlich auf Nutzerebene: Alle Daten eines Nutzers müssen auf dem selben Shard liegen.
- Der Schlüssel muss zu einer gleichmäßigen Verteilung führen
  - Beispiel: Wenn 80% der Nutzer aus dem selben Bundesstaat kommen, ist das ein ungeeigneter Schlüssel, da der Datenbank Cluster ungleichmäßig ausgelastet ist.
- **Kardinalität**: Möglichen Werte eines Schlüssel: Boolean Wert nur zwei Shards Max
- **Frequenz**: Verteilung der Werte über gesamten Datensatz

Bereichsbasiertes Sharding

- Verteilung über einen Wertebereich eines Schlüssels:
  - Anfangsbuchstabe Nachname: A-D,...

Name	Shard-Schlüssel
Beginnt mit A bis I	A
Beginnt mit J bis S	B
Beginnt mit T bis Z	C

Gehashtes Sharding

- Schlüssel wird mit Hilfe eines Hashes einer Zeile generiert
- **Pro:** Gleichmäßige Verteilung
- **Con:** Wenig Kontrolle über Datenlokalität – für viele Cases ungeeignet

Geo Sharding

- Verteilung der Daten nach geographischem Standort
- Pro: Besonders spannend für Systeme die sich über mehrere RZs und Kontinente erstrecken

Probleme von verteilten Systemen

- Umgang mit Teilausfällen von System (Wahrscheinlichkeit steigt mit Menge der Nodes) z.B. Vermeidung von Splitbrain Verfahren
- Konsistenz der Daten → später mehr
- Netzwerklatenzen → Georedundanzen lassen sich in vielen Systemen nicht ohne weiteres realisieren
- Komplexität in Architektur und Betrieb
- Rack Awareness bei VM & Containerbasierten Systemen
- uvm.

Weitere wichtige Begriffe

- Neben Sharding sind auch andere Faktoren bei der Skalierung von Datenbanken wichtig:
- Indexing
- Replication
- Caching
- Denormalisierung
- Einige der Konzepte werden im Laufe der Veranstaltung weiter beleuchtet

Verteilte Anfragen

- Verteilte Anfragen sind häufig nicht zu vermeiden
- Nicht jede Verteilte Anfrage ist zwangsläufig Sequentiell: siehe Map Reduce
- In den kommenden Vorlesungen werden wir lernen, wie verschiedene Systeme mit Sequentiellen Anfragen umgehen

Relation vs. Dokumentenorientiert

- Was wir gelernt haben: Viele NOSQL Datenbanken wie z.B. Dokumentenorientierte Datenbanken sind leichter zu skalieren als relationale Datenbanken
- Warum ist das so?

NOSQL Skalierbarkeit

- NOSQL Datenbanken speichern Daten denormalisiert:
  - Alle notwendigen Informationen sind in einem Dokument vorhanden
  - Keine Joins möglich, diese sind besonders ineffizient
  - Sharding ist erheblich einfacher
- BASE vs. ACID => siehe folgende Folien

ACID

Transaktionen

- Eine Transaktion ist eine Arbeitseinheit (oder auch operation)
- Wird entweder vollständig abgeschlossen oder garnicht
- Hinterlässt die Datenbank in einem **konsistenten Status**
- Wie beschreibt man eine solche Transaktion akademisch?

ACID

- Atomicity, Consistency, Isolation, Durability
- Acronym bestehend aus 4 Attributen, die gegeben sein müssen um die Gültigkeit der Daten trotz Fehlern, Stromausfällen und anderen Missgeschicken zu gewährleisten
- Datenbanken die über diese 4 Attribute verfügen nennt man „Transactional Systems“

Atomicity

- Jede Operation in einer Datenbank (Read, Write, Update, Delete) wird entweder vollständig oder gar nicht ausgeführt (im Englischen oft als Single Unit of Work bezeichnet).
- Diese Eigenschaft stellt sicher, dass keine Daten verloren gehen oder unvollständige oder beschädigte Daten geschrieben werden.
- Beispiel: Schreibprozess stürzt beim Schreiben ab: Alle Operationen werden rückgängig gemacht.

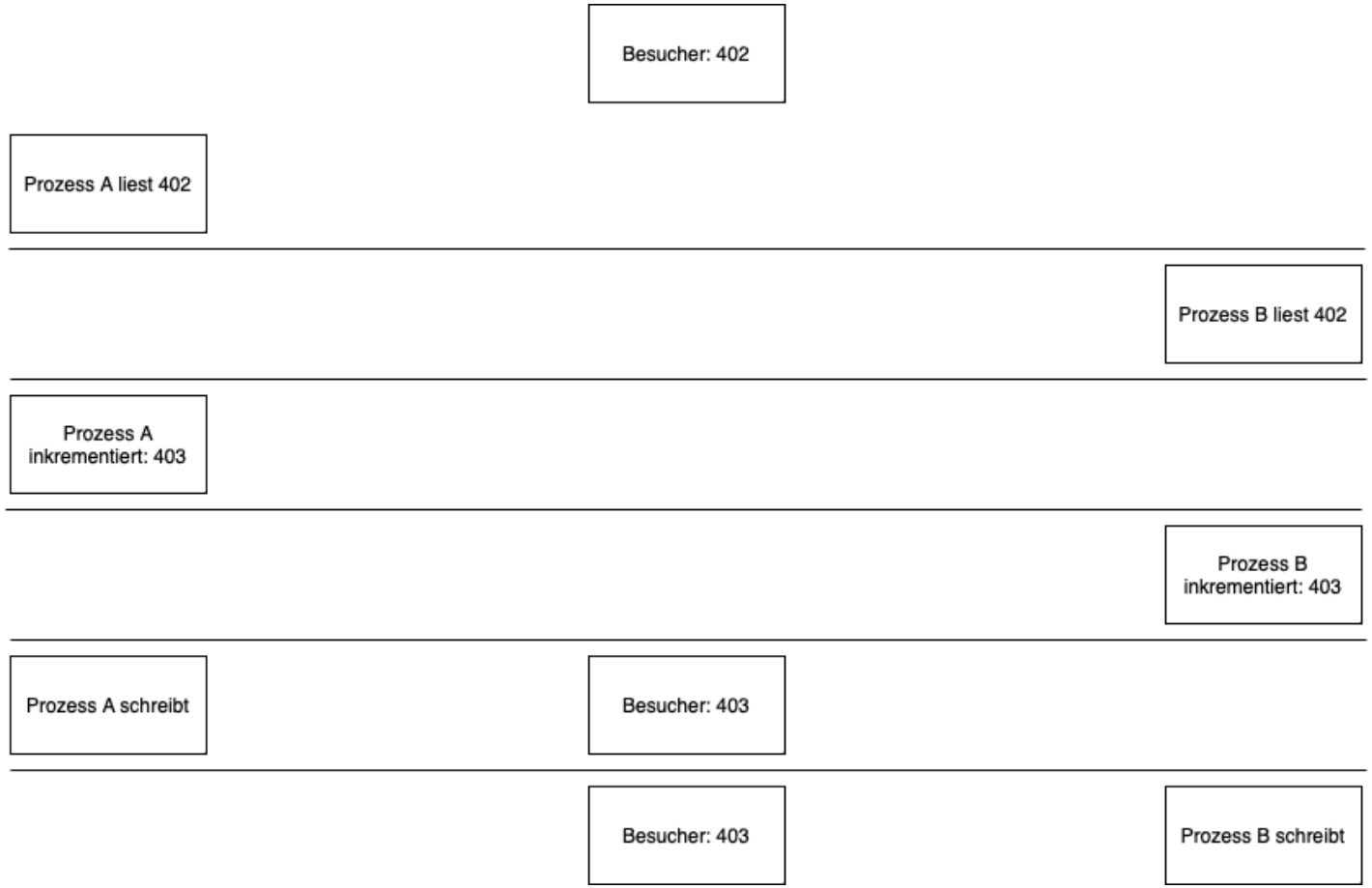
Consistency

- Die Transaktion hinterlässt nach der Operation einen konsistenten Datenbankzustand.
- Alle im Datenbankschema definierten Konsistenzbedingungen werden erzwungen (geschriebene Daten entsprechen dem definierten Schema).

Isolation

- Häufig greifen mehrere Prozesse gleichzeitig auf Datenbanken zu.
- Man stelle sich ein Kassensystem bei einem Konzert mit mehreren Eingängen vor.
  - Das System muss sowohl die Anzahl der Eintrittskarten als auch die Gesamtzahl der Besucher zählen.
  - Daher kann es vorkommen, dass mehrere Systeme gleichzeitig versuchen, den Wert zu aktualisieren: Der Aktualisierungsprozess sieht wie folgt aus: Zuerst wird die Anzahl der Besucher ausgelesen, dann im Speicher inkrementiert und schließlich der neue Wert in die Datenbank geschrieben.
  - Ohne Isolation kommt es unweigerlich dazu, dass sich die Prozesse in die Quere kommen.

Isolation


Besucher: 402
Prozess A liest 402
Prozess B liest 402
Prozess A
inkrementiert: 403
Prozess B
inkrementiert: 403
Prozess A schreibt
Besucher: 403
Besucher: 403
Prozess B schreibt

Isolation

- Das Isolationsprinzip verhindert die gegenseitige Beeinflussung mehrerer Prozesse.
- Realisiert wird dies durch Sperrverfahren, die die für die Transaktion notwendigen Daten sperren, für den Transaktionsablauf sperren.
- Nebenläufigkeit wird eingeschränkt
- Kann in den meisten Datenbanken konfiguriert werden, um die Leistung von Transaktionen auf Tabellen zu verbessern, die diese Funktionen nicht benötigen (beispielsweise Daten die Immutable sind).

Durability

- Daten sind nach Abschluss der Transaktion garantiert Dauerhaft gespeichert
- Speicherung muss auch nach Systemfehler garantiert sein
- Wird durch führen eines Transaktionslogs ermöglicht (Grundlegend wichtiges Konzept)

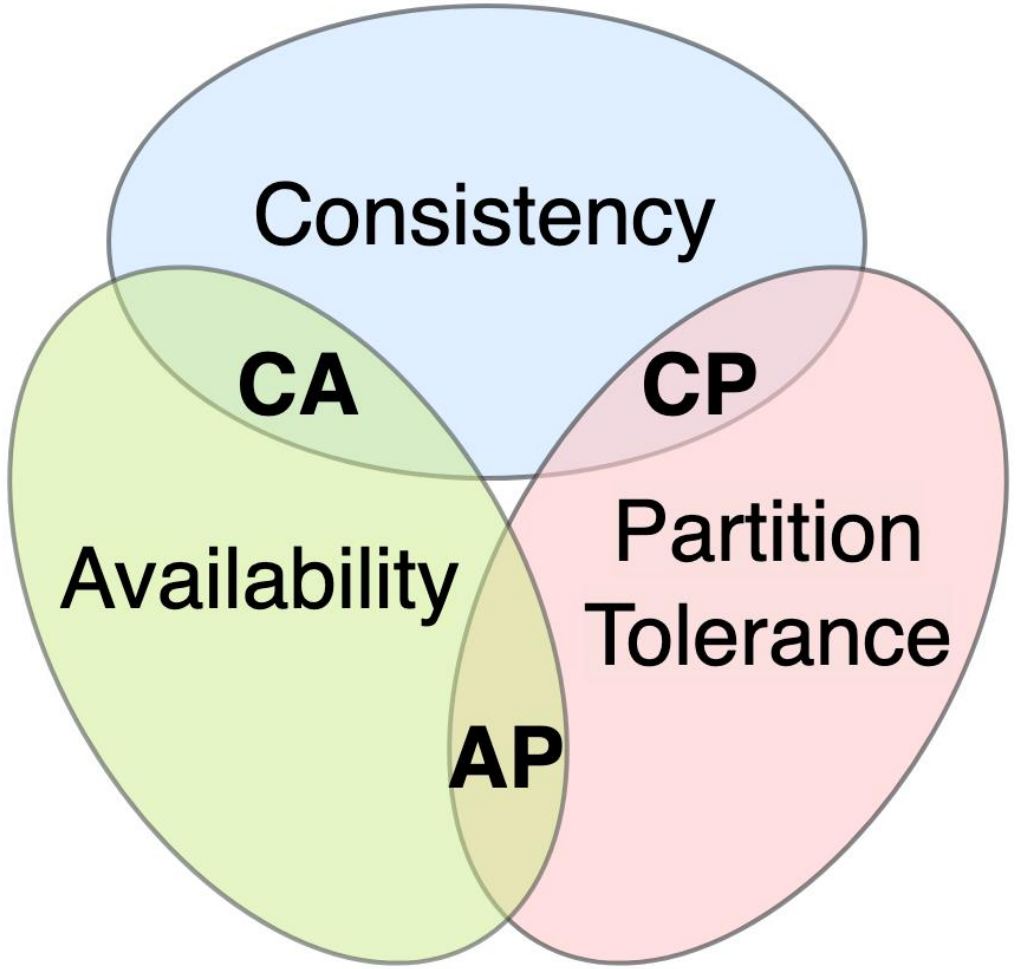
ACID

- Wird von fast allen relationalen Datenbanken unterstützt.
- Vorsicht, die meisten NOSQL-Datenbanken sind nicht ACID-kompatibel (dazu später mehr).
- Deutlich schwieriger in verteilten Systemen zu implementieren, insbesondere wenn Hochverfügbarkeit erreicht werden soll → CAP-Theorem

CAP

CAP Theorem

- Theorem in Datenbanktheorie
- Gilt für verteilte Datenbanken
- Balance zwischen: Konsistenz; Verfügbarkeit; Partition Toleranz
- Nur zwei der drei Eigenschaften können von einer verteilten Datenbank erreicht werden



CAP Theorem

- Bei der Auswahl von Datenbanksystemen für einen bestimmten Anwendungsfall ist das CAP-Theorem unbedingt zu berücksichtigen!
- Das Theorem setzt Partitionen voraus und gilt **nur** für verteilte Datenbanken
- Wo Datenbanken im CAP-Spektrum einzuordnen sind lässt sich nachschauen

Consistency

- Beschreibt das Verhalten einer Datenbank in Bezug auf die Aktualität der Daten bei einem Lesevorgang. Konsistent ist eine Datenbank, die immer die aktuellsten Daten zurückgibt.
- Ein System ist konsistent, wenn garantiert werden kann, dass eine Leseoperation L1, die zeitlich nach einer Schreiboperation S1 stattfindet, die Daten von S1 lesen kann.
- Es gibt verschiedene Stufen der Konsistenz (dazu später mehr)

Availability

- Jeder Request, an einen gesunden Datenbank Node muss zu einem Ergebnis führen
- Das Ergebnis muss nicht unbedingt das aktuellste sein ↔ Availibility bedeutet nicht Consistency

Partition Tolerance

- Das System funktioniert trotz Netzwerkfehler
- Beispielhafte Fehler: dropped partition, langsame Netzwerkverbindung zwischen Nodes, totalausfall des Netzwerks zwischen den Nodes

Gültigkeit des CAP Theorems

- Jedes verteilte Systeme ist der Gefahr von Netztwerkfehlern ausgesetzt
- Im Falle eines Fehlers gibt es zwei mögliche Verhalten:

Gültigkeit des CAP Theorems

1. **Consistency:** Das System kann nicht sicherstellen, dass die Daten konsistent sind (anderer Node kann eventuell neuere Daten halten): Es kommt zu einem Error oder Timeout
2. **Availability:** Das System verarbeitet die Anfrage mit dem Risiko, dass neuere Informationen auf dem nicht erreichbaren Datenbank-Note vorliegen

- Gibt es keine Partition-Tolerance kann sowohl Availability, als auch Consistency eingehalten werden
- ⇒ Datenbanken, die ACID implementieren implementieren Consistency, **Eventually Consistant** Datenbanken implementieren Availability

BASE

BASE

- Akronym und Hinweis auf chemische Äquivalente (Säuren und Basen)
- Gegenteil von ACID
- In NOSQL-Datenbanken verbreitetes Modell
- Steht für
  - Basically available
  - Soft state
  - Eventually consistent

Basically Available

- Benutzer können jederzeit gleichzeitig zugreifen
- Keine Sperrmechanismen → keine Wartezeiten bei gleichzeitigem Zugriff
- Hat Vorteile bei der Performance, führt aber zu mangelnder Consistency

Soft State

- Daten können sich in einem temporären Zustand befinden, der sich im Laufe der Zeit ändert.
- Übergangszustand eines Datensatzes, wenn er von mehreren Anwendungen gleichzeitig aktualisiert wird.
- Der Wert ist finalisiert, wenn alle Transaktionen abgeschlossen sind (siehe Eventual Consistency).
- Wert ändert sich, ohne dass eine weitere Transaktion oder ein Trigger ausgelöst wird

Eventually Consistent

- Der Datensatz ist konsistent, wenn alle Transaktionen abgeschlossen sind.
- Ab diesem Zeitpunkt sehen alle Anwendungen, die den Datensatz abfragen, den gleichen Wert.
- Das bedeutet aber auch, dass es Situationen gibt, in denen die Datenbank inkonsistent ist.
- Je nach Anwendungsfall kann dies ein Problem darstellen!