

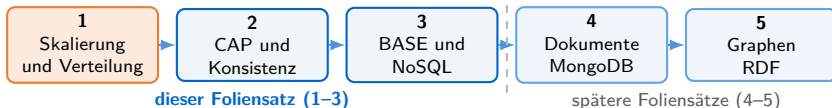
Datenbanken und Informationssysteme

8. Alternative Datenmodelle und No-SQL

Prof. Dr. Stefan Decker

📍 RWTH Aachen

Leitfrage: Welche Datenmodelle und Systemgarantien passen, wenn Daten verteilt, verschachtelt, dokumentorientiert oder graphförmig sind?



Roter Faden

- ▶ Verteilung schafft Skalierung, aber auch Koordinationsprobleme.
- ▶ CAP und BASE machen die Trade-offs sichtbar.
- ▶ NoSQL-Familien wählen bewusst andere Modellierungsformen.

Prüffrage für jedes Modell

- ▶ Welche Struktur ist natürlich?
- ▶ Welche Anfragen sind zentral?
- ▶ Welche Garantien werden stärker oder schwächer?

Abschnitt 1: Skalierung und Verteilung



Von einem Knoten zu vielen

Wie skalieren wir Datenbanksysteme, ohne Konsistenz und Verfügbarkeit zu verlieren?

Bisher

- ▶ RDBMS liefern ACID: typischerweise auf *einem* Knoten.
- ▶ Neue Lastprofile fordern Verteilung und Skalierung.

Jetzt: der Weg durch Abschnitt 1

- ▶ Motivation, Skalierung (vertikal/horizontal), Sharding.
- ▶ Amdahl-Grenzen, Replikation, Konsistenz, Koordination (2PC).

Roter Faden: Skalierung führt zu Verteilung; Verteilung erzwingt bewusste Trade-offs.

Motivation – warum Alternativen zu klassischen RDBMS?

RDBMS: starke Basis

- ▶ **Schema & Integrität:** klare Strukturen, Constraints
- ▶ **ACID:** verlässliche Transaktionen, klassisch auf *einem* Knoten
- ▶ **SQL:** deklarativ, standardisiert, optimierbar
- ▶ **Reife Technologie:** Indizes, Optimizer, Betrieb

⇒ **Ideal für** konsistenzkritische Anwendungen mit gut strukturierten Daten.

Neue Lastprofile

- ▶ **Volume/Velocity:** Datenmenge und Schreiblast wachsen stark
- ▶ **Variety:** JSON, RDF, Graphen, unstrukturierte Daten
- ▶ **Availability:** 24/7-Betrieb, Ausfälle tolerieren
- ▶ **Scale-out:** viele Standardserver statt ein immer größerer Server

⇒ **Druckpunkt:** Flexibilität (Schema!) und Verteilung werden wichtiger.

Kernidee: Nicht Ersatz-Hype, sondern passende Trade-offs für Workload und Zugriffsmuster: andere Systemarchitekturen.

Skalierungsstrategien – vertikal vs. horizontal

Ziel: Lastwachstum abfangen: mehr Leistung pro Server oder mehr Server im Verbund.

Vertikal: Scale-up



- ▶ **Prinzip:** CPU, RAM, Disk eines Knotens erhöhen
- + einfache Verwaltung, wenig Softwareanpassung
- + Konsistenz bleibt lokal einfacher
- teuer, technische Grenzen, Single Point of Failure

Horizontal: Scale-out



- ▶ **Prinzip:** Last und Daten auf mehrere Knoten verteilen
- + hohe Skalierbarkeit mit Standardhardware
- + Redundanz ermöglicht höhere Verfügbarkeit
- Konsistenz, Netzwerk und Betrieb werden schwieriger

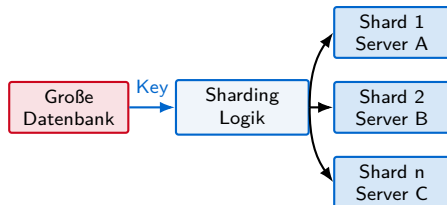
Fazit: Scale-out hilft bei Web-Scale-Lasten, bringt aber Sharding, Replikation und Konsistenz ins Spiel.

Datenverteilung durch Sharding

Mechanismus

Sharding zerlegt eine große Datenbank horizontal in kleinere, logische Teile: **Shards**.

- ▶ Jeder Shard liegt typischerweise auf einem eigenen Server.
- ▶ Ein **Sharding Key** entscheidet, welcher Datensatz wohin kommt.
- ▶ Häufige Regel: $\text{Shard} = f(\text{Key})$, z.B. per Hash oder Bereich.
- ▶ **Ziel:** Last verteilen und durch neue Server skalieren.



Trade-off: mehr Skalierbarkeit, aber neue Fragen zu Key-Wahl, shard-übergreifenden Abfragen, Konsistenz und Re-Sharding.

Sharding als Parallelisierungsmuster

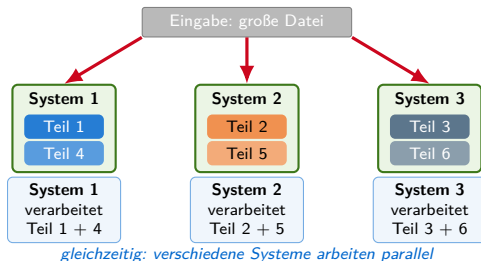
Beschleunigung und Grenzen

Was wird schneller?

- ▶ Shard-lokale Anfragen gehen direkt zum passenden Server.
- ▶ Unabhängige Partitionen lassen sich parallel verarbeiten.

Was begrenzt den Gewinn?

- ▶ **Hot Shards:** ein schlechter Key erzeugt Lastspitzen.
- ▶ **Cross-Shard-Operationen:** Joins, Aggregationen und Transaktionen brauchen Koordination.
- ▶ **Re-Sharding:** wachsende Systeme müssen Daten neu verteilen.



Überleitung: Wie viel schneller wird es wirklich? Der nicht parallelisierbare Anteil setzt die Grenze.

Grenzen der Parallelisierung: Amdahlsches Gesetz

Gene Amdahl (1967): Grenze für Speedup durch Parallelisierung.

Frage: Wie groß kann $S(p)$ maximal werden, wenn Arbeit auf p Einheiten verteilt wird?

Speedup nach Amdahl

$$S(p) = \frac{t}{t_p} = \frac{t}{t \cdot s + t \cdot \frac{1-s}{p}} = \frac{1}{s + \frac{1-s}{p}}$$

Bedeutung der Terme

- ▶ p : Anzahl paralleler Einheiten
- ▶ t : sequentielle Ausführungszeit
- ▶ t_p : Ausführungszeit auf p Einheiten
- ▶ s : Anteil, der sequentiell bleiben muss
- ▶ $1 - s$: parallelisierbarer Anteil

Kernaussage: Der sequentielle Anteil s ist die harte Grenze: $\lim_{p \rightarrow \infty} S(p) = 1/s$. Mehr Knoten helfen nur dem parallelisierbaren Anteil.

Amdahlsches Gesetz: Beispielrechnung mit Realitätscheck

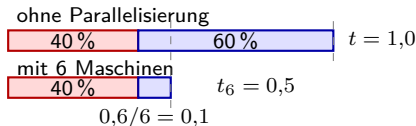
Annahmen

- ▶ 60 % der Anfragebearbeitung ist parallelisierbar.
- ▶ 40 % bleibt sequentiell.
- ▶ Die Tupelselektion nutzt $p = 6$ Maschinen.

Merke: Die 6 Maschinen teilen nur den 60 %-Anteil; der 40 %-Rest bleibt unvermeidbar.

Maximaler Speedup

$$S(6) = \frac{1}{s + \frac{1-s}{p}} = \frac{1}{0,4 + \frac{0,6}{6}} = 2,0$$



Realitätscheck: $S(6) = 2,0$ ist eine ideale Obergrenze. In DB-Clustern senken Kommunikation, Synchronisation, Hot Shards und Cross-Shard-Operationen den realen Speedup weiter.

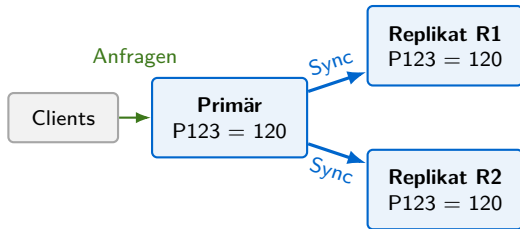
Datenverfügbarkeit und Fehlertoleranz: Replikation

Replikation: dieselben Daten werden mehrfach gespeichert. Sharding teilt Daten auf; Replikation vervielfacht sie.

Konsistenzfrage: Wie bleiben Kopien bei Updates auf demselben Stand?

Warum replizieren?

- ▶ **Verfügbarkeit:** Bei Ausfall bleibt eine Kopie erreichbar.
- ▶ **Leseskalierung:** Leseanfragen lassen sich verteilen.
- ▶ **Niedrige Latenz:** Kopien können näher an Nutzern liegen.

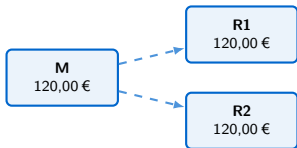


Trade-off: Mehr Kopien erhöhen Verfügbarkeit; **Konsistenz** erfordert, dass jede Änderung die Replikate erreicht.

Replikation: Von Konsistenz zu Inkonsistenz

Szenario: Produkt P123 kostet 120,00 € und soll auf 99,95 € gesetzt werden. Master M , Replikate $R1$, $R2$. Wir betrachten drei Zeitpunkte.

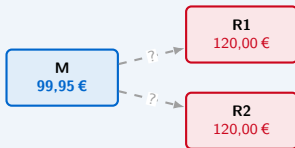
1. Vor dem Update



✓ **Konsistent**

Alle Kopien zeigen denselben Wert. Reads sind eindeutig.

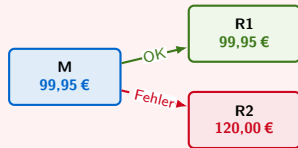
2. Update am Master



△ **Konsistenzfenster**

Replikation läuft, aber noch nicht abgeschlossen. Reads können alt oder neu sein.

3. Partielle Propagierung



× **Manifest inkonsistent**

Update war *nicht atomar* über alle Knoten: $M, R1 \neq R2$.

Beobachtung: Naive Replikation kennt zwei Fehlerquellen: **(1)** ein *Konsistenzfenster*, während Updates noch unterwegs sind, und **(2)** *partielle Fehler*, bei denen einzelne Replikate dauerhaft zurückbleiben.

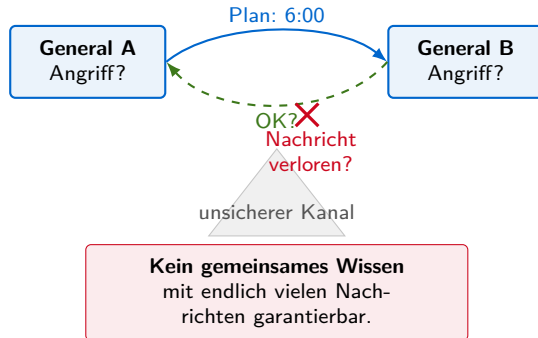
⇒ **Motivation:** Verteilte atomare Updates benötigen Koordinationsprotokoll: **Two-Phase Commit (2PC)**.

Zwei Generäle: Grenzen verteilter Einigung

Kernfrage: Ist gemeinsame Entscheidung garantierbar, wenn Nachrichten beliebig verloren gehen können?

Warum ACKs nicht reichen

- ▶ Endliche Protokolle haben immer eine letzte Bestätigung.
- ▶ Deren Sender weiß nicht, ob sie ankommt.
- ▶ Kein garantiertes gemeinsames Wissen: „Wir handeln beide.“
- ▶ **FLP-Bezug:** In asynchronen Systemen sind Verzögerung und Ausfall nicht sicher unterscheidbar.



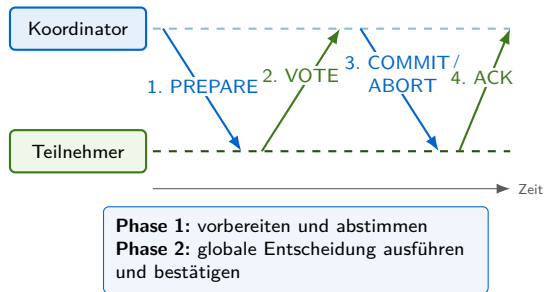
Konsequenz: Das allgemeine Einigungsproblem über beliebig unzuverlässige Kanäle ist mit endlich vielen Nachrichten nicht lösbar. Das folgende Koordinationsverfahren funktioniert daher nur in einem engeren Modell mit expliziten Annahmen über Nachrichten, Ausfälle und Wiederanlauf.

Two-Phase Commit (2PC): Verteilte Atomarität

Ziel: Eine verteilte Transaktion wird auf allen beteiligten Systemen **committed** oder auf allen **abgebrochen**: Alles oder nichts — **ACID-Atomarität** (Kap. 7), jetzt über Knotengrenzen hinweg.

Ausgangsproblem

- ▶ Einzelne Knoten können unterschiedlich weit sein.
- ▶ Partieller Erfolg erzeugt Inkonsistenz.
- ▶ Ein Koordinator sammelt Stimmen und trifft die globale Entscheidung.



2PC $\neq$ 2PL: *Commit* koordiniert *eine* Transaktion atomar über *mehrere* Knoten; *Locking* (2PL, Kap. 7) synchronisiert *nebenläufige* Transaktionen auf *einem* Knoten.

2PC Phase 1: Prepare

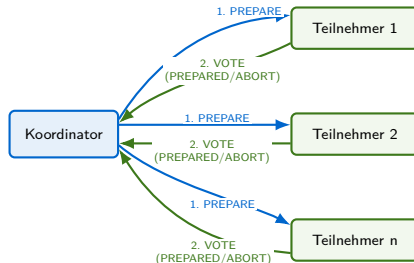
Ziel: Commit-Bereitschaft **aller** Teilnehmer prüfen. **PREPARE:** vorbereiten, aber auf das GO des Koordinators warten.

Ablauf der Prepare-Phase

Koordinator → alle Teilnehmer: **PREPARE**-Anfrage.

Jeder Teilnehmer lokal: Operationen ausführen (ohne finales Commit); Änderungen und **Prepared**-Status persistent loggen; Ressourcen sperren; lokale Commit-Fähigkeit prüfen.

Teilnehmer → **Koordinator**: **VOTE**: **PREPARED** oder **ABORT**.



PREPARED (JA)

Verbindliche Zusage: bereit, wartet auf finale Anweisung; blockiert, kein einseitiger Rollback.

ABORT (NEIN)

Kann/will nicht committen; lokaler Rollback; Ressourcen frei.

Kernpunkt

PREPARED ist bindend: Warten auf **COMMIT** oder **ABORT**.

2PC Phase 2: Commit/Abort

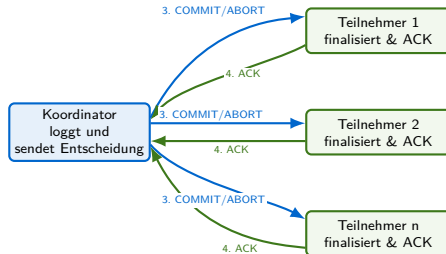
Ziel: Der Koordinator trifft eine globale Entscheidung und setzt sie bei allen Teilnehmern durch.

Entscheidung

- ▶ **Alle PREPARED $\Rightarrow$ COMMIT**
- ▶ **ABORT** oder Timeout $\Rightarrow$ **ABORT**
- ▶ Entscheidung persistent loggen

Umsetzung

- ▶ Entscheidung senden: **3. COMMIT/ABORT**
- ▶ Finalisieren oder zurückrollen
- ▶ Ressourcen frei, danach **4. ACK**



Merke: Finale Nachricht verloren?
PREPARED-Teilnehmer blockieren.

2PC: Vorteile und Grenzen

Einordnung: Two-Phase Commit tauscht verteilte Atomarität gegen synchrones Warten.

Was 2PC leistet

- ▶ **Alles-oder-Nichts** über Teilnehmer hinweg
- ▶ Einheitlicher Endzustand: commit oder abort
- ▶ Grundlage für strikte Konsistenz

Was 2PC kostet

- ▶ **Blockierung** nach **PREPARED**
- ▶ Ressourcen bleiben gesperrt
- ▶ Mehr Nachrichten und synchrones Warten
- ▶ Koordinator als Engpass/SPOF

Zwischenfazit: 2PC ist stark, wenn Atomarität wichtiger ist als Verfügbarkeit.
CAP, BASE und NoSQL beschreiben die alternativen Kompromisse.



Konsistenz, Verfügbarkeit, Partitionen

Was kann ein verteiltes System im Fehlerfall noch garantieren?

Bisher

- ▶ Sharding verteilt Daten auf mehrere Knoten.
- ▶ Replikation erhöht Verfügbarkeit.
- ▶ 2PC koordiniert atomare Entscheidungen, kann aber blockieren.

Jetzt

- ▶ Netzwerkpartitionen machen Garantien explizit sichtbar.
- ▶ CAP beschreibt den Konflikt zwischen **C**, **A** und **P**.
- ▶ Die Wahl der Garantie prägt die späteren NoSQL-Designs.

Nächster Schritt: CAP macht die Trade-offs präzise; BASE und NoSQL zeigen danach typische Designantworten.

Das CAP-Theorem: Herkunft und Aussage

Idee: In verteilten Systemen stehen starke Garantien unter Netzwerkfehlern in einem fundamentalen Konflikt.

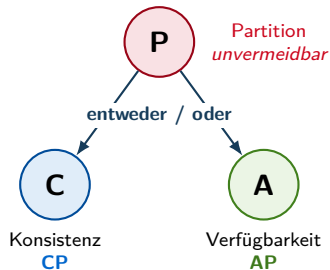
Herkunft

- ▶ **2000:** Brewer formuliert die CAP-Vermutung.
- ▶ **2002:** Gilbert & Lynch liefern den Beweis.
- ▶ **Kontext:** skalierbare verteilte Internet-Dienste.

Kernaussage

Bei Partition P kann ein System nicht gleichzeitig garantieren:

- ▶ **C:** alle sehen denselben aktuellen Zustand
- ▶ **A:** erreichbare Knoten antworten immer



Präziser: nicht gleichzeitig *Konsistenz der Daten* und *Verfügbarkeit* für jede Anfrage bei Partitionierung des Netzwerks.

Quelle: E. Brewer, *CAP Twelve Years Later*, IEEE Computer, 2012.

Die drei CAP-Eigenschaften

C – Consistency

Konsistenz im CAP-Sinn

- ▶ Jeder Read sieht den letzten Write oder bekommt einen Fehler.
- ▶ Replikas wirken wie eine einzelne aktuelle Kopie.

A – Availability

Verfügbarkeit

- ▶ Jede Anfrage an einen funktionierenden Knoten erhält Antwort.
- ▶ Antwort ist nicht zwingend die aktuellste Version.

P – Partition Tolerance

Partitionstoleranz

- ▶ Das System rechnet mit Nachrichtenverlust oder Netzwerksplits.
- ▶ Teilgruppen müssen kontrolliert weiterarbeiten.

Wichtig: CAP-Konsistenz meint die Sicht auf den aktuellsten Wert im verteilten System. Das ist nicht dasselbe wie das **C** in ACID.

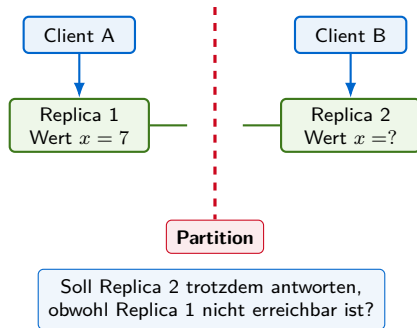
Praxisblick: In großen verteilten Systemen ist *P* kein Luxus, sondern ein Fehlerfall, mit dem das Design umgehen muss.

Warum CAP? Der Fehlerfall zählt

CAP wird relevant, sobald eine Netzwerkpartition auftritt: Dann können Replikas nicht mehr sicher koordinieren.

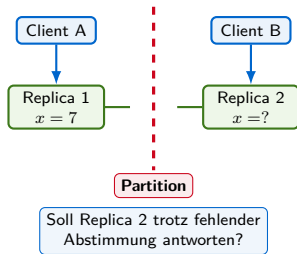
Entscheidung im Fehlerfall

- ▶ Keine Nachrichten zwischen den Teilnetzen.
- ▶ Clients erreichen unterschiedliche Replikas.
- ▶ **Antworten:** Verfügbarkeit bleibt, Konsistenz kann leiden.
- ▶ **Warten/ablehnen:** Konsistenz bleibt, Verfügbarkeit sinkt.



CAP im Fehlerfall: CP oder AP?

Bei Netzwerkpartitionen entscheidet CAP: Ohne sichere Koordination priorisiert das System C (Konsistenz) oder A (Verfügbarkeit).



CP: Konsistenz

- ▶ Leader oder Quorum antwortet.
- ▶ Unsichere Anfragen blockieren oder Fehler liefern.
- ▶ Keine widersprüchlichen Writes.

Typisch: Konten, Locks.

AP: Verfügbarkeit

- ▶ Jede erreichbare Partition antwortet weiter.
- ▶ Lokale Reads/Writes werden akzeptiert.
- ▶ Divergenzen später auflösen.

Typisch: Warenkorb, Feeds.

Merke: Bei P kann ein System nicht gleichzeitig volle C und volle A garantieren: Es muss im Fehlerfall entscheiden.

CAP-Konsistenz vs. ACID-Konsistenz

Achtung: “Konsistenz” bedeutet in ACID und CAP nicht dasselbe.

ACID-Konsistenz: gültiger Zustand

- ▶ Transaktionen erhalten die definierten Invarianten.
- ▶ Constraints, Referenzen und Anwendungsregeln bleiben erfüllt.
- ▶ Fokus: logische Korrektheit innerhalb einer Transaktion.

CAP-Konsistenz: aktuelle Sicht

- ▶ Reads sehen den aktuellsten globalen Write oder schlagen fehl.
- ▶ Replikas wirken wie eine einzige aktuelle Kopie.
- ▶ Fokus: Sichtbarkeit und Gleichstand verteilter Datenkopien.

Merke: ACID-Konsistenz fragt: “Ist der Zustand regelkonform?” CAP-Konsistenz fragt: “Sehen alle denselben aktuellen Wert?”

CAP-Fazit: Warum BASE?

Wenn unter Partition Verfügbarkeit wichtiger ist, wird strikte Konsistenz oft bewusst gelockert.

Ausgangspunkt: CAP

- ▶ P bleibt in verteilten Systemen praktisch relevant.
- ▶ AP-Systeme antworten auch bei gestörter Kommunikation.
- ▶ Anwendungen müssen temporär unterschiedliche Werte tolerieren.

Nächste Fragen

- ▶ Welche Garantien bleiben erhalten?
- ▶ Wie beschreibt man “weiche” Zustände?
- ▶ Wann werden Replikas wieder gleich?

CAP-Trade-off

Designprinzipien

BASE-Modell

BASE: Basically Available, Soft state, Eventually consistent



Von strikten Garantien zu bewussten Trade-offs

Warum viele verteilte Systeme Konsistenz anders organisieren

Bisher

- ▶ ACID steht für starke Transaktionsgarantien.
- ▶ CAP zeigt Grenzen unter Netzwerkpartitionen.
- ▶ Strikte Koordination kann Latenz und Blockierung erzeugen.

Jetzt

- ▶ BASE beschreibt schwächere, aber skalierbare Garantien.
- ▶ NoSQL ist eine Systemfamilie, kein einzelnes Datenmodell.
- ▶ Die Modellwahl folgt Zugriffsmustern und Verteilungsanforderungen.

Nächster Schritt: Erst die Systemgarantien klären, dann die NoSQL-Familien und ihre Datenmodelle vergleichen.

BASE und NoSQL: Agenda

1. Motivation

- ▶ CAP und 2PC zeigen Grenzen strikter Koordination.
- ▶ Verteilte Systeme priorisieren oft Verfügbarkeit und Skalierung.

2. BASE-Modell

- ▶ Basically Available, Soft state, Eventually consistent.
- ▶ Gemeinsame Sprache für optimistischere Garantien.

3. BASE-Systeme

- ▶ Eigenschaften und praktische Implikationen.
- ▶ Umgang mit Replikation, Ausfällen und Konvergenz.

4. NoSQL-Bewegung

- ▶ Motivation und Abgrenzung zu klassischen RDBMS.
- ▶ Schemaflexibilität, horizontale Skalierung, Kategorien.

Rahmen: ACID → CAP-Trade-offs → BASE → NoSQL-Systemfamilien

Das BASE-Modell: Alternative zu ACID

Ausgangspunkt CAP: AP-Systeme lockern strikte Konsistenz, um trotz Partitionen verfügbar und skalierbar zu bleiben. **BASE** liefert dafür das Designvokabular.

ACID: pessimistisch

- ▶ Konsistenz vor der Freigabe erzwungen.
- ▶ Koordination, Locks, Commit-Protokolle.
- ▶ Verteilt oft teuer für Verfügbarkeit und Skalierung.

BASE: optimistisch

- ▶ Priorität: Verfügbarkeit und Skalierbarkeit.
- ▶ Temporär divergierende Replikas sind erlaubt.
- ▶ System konvergiert später wieder.

BASE = Basically Available, Soft State, Eventually Consistent

Basically Available

- ▶ System antwortet auch bei Teilausfall.
- ▶ Antwort kann veraltet, partiell, degradiert sein.
- ▶ Verfügbarkeit *vor* sofortiger Konsistenz.

Soft State

- ▶ Zustandsänderung *ohne* neues Nutzerupdate.
- ▶ Ursachen: Replikation, Caches, Reparaturen.
- ▶ Keine permanente globale Konsistenz.

Eventually Consistent

- ▶ Ohne neue Updates konvergieren alle Replikas.
- ▶ Temporäre Inkonsistenz wird akzeptiert.
- ▶ *Stale reads* sind möglich.

Kernidee: verfügbar bleiben, Abweichungen tolerieren, später wieder stabilisieren.

Konsistenzmodelle (nach Vogels 2009)

Kontext: "Eventual consistency" ist ein Oberbegriff. Praktische Systeme wählen oft stärkere, client-zentrierte Garantien.

Modell	Garantie / Intuition
Causal consistency	Kausale Abhängigkeiten bleiben sichtbar: Wer eine Antwort sieht, sieht auch den auslösenden Post.
Read-your-writes	Nach einem eigenen Update sieht derselbe Prozess seinen neuen Wert und fällt nicht auf einen alten Stand zurück.
Session consistency	Read-your-writes gilt innerhalb einer Sitzung; nach Sitzungswechsel kann die Garantie schwächer sein.
Monotonic reads	Hat ein Prozess Wert X gesehen, liefern spätere Reads keinen älteren Zustand mehr.
Monotonic writes	Schreibvorgänge desselben Prozesses werden in seiner Reihenfolge serialisiert.

Merke: Diese Modelle machen AP-Systeme programmierbarer, ohne globale strikte Konsistenz zu verlangen.

BASE-Systeme: Eigenschaften und Implikationen

BASE-Philosophie: weniger globale Koordination, mehr Verfügbarkeit bei Last und Ausfällen.

Systemdesign

- ▶ Priorität: Verfügbarkeit (A), Partitionstoleranz (P), Skalierung.
- ▶ Updates laufen oft asynchron, ohne globales 2PC-Commit.
- ▶ Optimistisch: Konflikte und temporäre Inkonsistenz sind später lösbar.

Anwendung und UX

- ▶ Anwendungslogik muss stale reads und Konflikte behandeln.
- ▶ Komplexität wandert teilweise von der DB in die Anwendung.
- ▶ Nutzer sehen hohe Verfügbarkeit, aber nicht immer den neuesten Stand.

Datenmodell: häufig flexibel oder schema-less; das begünstigt schnelle Anpassungen und agile Entwicklung.

NoSQL: Motivation und Merkmale

NoSQL: historisch oft als “No SQL” verstanden; später auch als “Not Only SQL” umgedeutet. Gemeint ist ein breites Spektrum jenseits klassischer RDBMS, meist als Ergänzung statt als pauschaler Ersatz.

Kern-Motivationen

- ▶ **Scale-out:** horizontale Skalierung für Big Data und hohe Last.
- ▶ **Uptime:** Verfügbarkeit und Partitionstoleranz, oft AP-orientiert.
- ▶ **Flexibilität:** variable oder semistrukturierte Daten.
- ▶ **Performance:** Optimierung für spezielle Workloads.
- ▶ **Kosten:** Cluster aus Standard-Hardware.

Typische Merkmale

- ▶ Nicht-relationales Datenmodell.
- ▶ Verteiltes, clusterfähiges Design.
- ▶ Hohe Schema-Flexibilität.
- ▶ Häufig gelockerte Konsistenzmodelle.
- ▶ Spezifische APIs und Abfragesprachen, nicht nur SQL.

Wichtig: NoSQL ist ein breites Technologiespektrum, kein einzelner Standard.

NoSQL-Hauptkategorien im Überblick

Key-Value Stores

- ▶ **Prinzip:** Schlüssel → Wert, oft opaker Blob.
- ▶ **Stärke/Beispiele:** schnelle Lookups; Redis, Memcached, DynamoDB.

Wide-Column Stores

- ▶ **Prinzip:** Spaltenfamilien und spärlich besetzte Zeilen.
- ▶ **Stärke/Beispiele:** Scale-out; Cassandra, HBase, Bigtable.

Dokumentendatenbanken

- ▶ **Prinzip:** flexible Dokumente, z. B. JSON oder BSON.
- ▶ **Stärke/Beispiele:** Schemaflexibilität; MongoDB, Couchbase.

Graphdatenbanken

- ▶ **Prinzip:** Knoten, Kanten, Properties oder RDF-Tripel.
- ▶ **Stärke/Beispiele:** Beziehungen; Neo4j, Neptune, Virtuoso.

Wichtig: Grenzen sind oft fließend; viele Systeme kombinieren mehrere Modelle.
Im Folgenden behandeln wir Dokumente und Graphen exemplarisch.

CAP

- ▶ Partitionen erzwingen Trade-offs.
- ▶ AP-Systeme lockern strikte Konsistenz.

BASE

- ▶ Verfügbar bleiben.
- ▶ Soft state akzeptieren.
- ▶ Später konvergieren.

NoSQL

- ▶ Nicht nur relationale Modelle.
- ▶ Schemaflexibel und horizontal skalierbar.
- ▶ Mehrere Systemfamilien.

Kernbotschaft: Moderne Datenbanksysteme werden nach Anforderung gewählt: Konsistenz, Verfügbarkeit, Skalierung, Datenmodell und Workload müssen zusammenpassen.

Ausblick: Dokumentendatenbanken am Beispiel MongoDB und Graphdatenbanken mit RDF und Semantic Web.

[Amdahl, 1967]

Amdahl, G.M.: Validity of the single processor approach to achieving large scale computing capabilities. AFIPS Spring Joint Computer Conf., 483–485 (1967).

[Brewer, 2000]

Brewer, E.A.: Towards robust distributed systems (abstract). PODC, Portland, Oregon, USA, 7 (2000).

[Brewer, 2012]

Brewer, E. (2012). CAP twelve years later: How the “rules” have changed. Computer, 45(2), 23–29.

[Elmasri & Navathe, 2017]

Elmasri, R., & Navathe, S. (2017). Fundamentals of Database Systems (7th ed.). Pearson.

[Fischer et al., 1985]

Fischer, M.J., Lynch, N.A., & Paterson, M.S.: Impossibility of distributed consensus with one faulty process. Journal of the ACM, 32(2), 374–382 (1985).

[Gilbert & Lynch, 2002]

Gilbert, S., & Lynch, N.: Brewer’s conjecture and the feasibility of consistent, available, partition-tolerant web services. ACM SIGACT News, 33(2), 51–59 (2002).

[Lourenço et al., 2015]

Lourenço, J.R., Cabral, B., Carreiro, P., Vieira, M., & Bernardino, J. (2015). Choosing the right NoSQL database for the job. Journal of Big Data, 2(1), 1–26.

[McMurtry et al., 2013]

McMurtry, D. et al.: Data Access for Highly-Scalable Solutions: Using SQL, NoSQL, and Polyglot Persistence. Microsoft, 2013.

[Marz & Warren, 2015]

Marz, N., Warren, J.: Big Data: Principles and Best Practices of Scalable Real-Time Data Systems. Manning, 2015.

[Raj & Deka, 2018]

Raj, P., Deka, G.C.: A Deep Dive into NoSQL Databases: The Use Cases and Applications. Elsevier Science & Technology, 2018.

[Sadalage & Fowler, 2012]

Sadalage, P.J., Fowler, M.: NoSQL Distilled. Addison-Wesley, 2012.

[Vogels, 2009]

Vogels, W.: Eventually Consistent. Communications of the ACM, 52(1), 40–44, 2009.