

Daten & Serialisierung + DIKW + Vergleichsachsen Teaser

Willkommen zu „Databases Unlocked“ – einer strukturierten Reise durch die Evolutionsstufen der Datenspeicherung! Heute starten wir bewusst „unten“ bei rohen Datenformaten, um zu verstehen, warum heutige Systeme so gestaltet sind, wie sie sind.

Was Sie heute erwartet:

Diese Vorlesung ist **kein nostalgischer Rückblick**, sondern die **Begründung** dafür, warum moderne Datenbanksysteme so funktionieren, wie sie funktionieren.

Lassen Sie uns die Reise durch die Datenspeicher-Paradigmen skizzieren – von primitiv bis hochentwickelt. Wir durchlaufen sieben Blöcke, wobei jeder Block auf den Schwächen des vorherigen aufbaut. Block 1 zeigt uns die rohe Realität: Serialisierungsformate sind einfach, aber fehleranfällig. Die Blöcke 2 bis 6 führen uns durch verschiedene Paradigmen – vom kompakten Paradigmen-Überblick über den relationalen Kern (Algebra, SQL, Performance) bis zu fortgeschrittenen Konzepten. Block 7 vereint schließlich Graph-Datenbanken und polyglotte Architekturen mit verteilten Systemen.

Unsere Reise durch die Datenspeicher-Evolution

Block 1: Die rohe Realität

-  **Daten & Serialisierung** ← Heute: L1
-  **Paradigmen-Überblick** (L2–L6: KV, Document, Wide Column, Column, Trade-offs)

Block 2: Relationale Grundlagen

-  **Relationale Algebra & SQL Basics** (L7–L9 + Exercises)
-  **FROM σ , π , \bowtie TO SELECT, JOIN, CTE**

Block 3: Relationale Integrität

-  **Normalisierung & Constraints** (L10–L11)
-  **Transaktionen & ACID** (L12 + Exercise)

Block 4: Performance & Optimierung

-  **Indexe & Query-Optimierung** (L13–L15 + Exercise)
-  **B-Trees, EXPLAIN, Materialized Views**

Block 5: Fortgeschrittene Konzepte

-  **Locking vs. MVCC** (L16)
-  **Views, Triggers, Stored Procedures** (L17)

Block 6: Graph & Polyglot

-  **Graph Databases** (L18 + Exercise)
-  **Property Graphs, Traversal, Pattern Matching**

Block 7: Polyglot & Verteilung

-  **Polyglot Persistence** (L19: CQRS, Event Sourcing)
-  **Verteilte Systeme** (L20–L21: Replikation, CAP, Konsistenz)

Begleitend durchlaufen Sie ein Mini-Projekt mit dokumentierten Designentscheidungen – von rohen CSV-Dateien bis zur polyglotten Architektur. Diese vier Meilensteine sind keine theoretischen Übungen, sondern praktische Erfahrungen mit echten Trade-offs. Meilenstein 1 konfrontiert Sie mit den Limitationen flacher Dateien. Meilenstein 2 zwingt Sie zur Schema-Evolution – was passiert, wenn sich Anforderungen ändern? Meilenstein 3 bringt Performance ins Spiel: Wann lohnt sich ein Index wirklich? Und Meilenstein 4 stellt die große Frage: Wann rechtfertigt Normalisierung ihren Aufwand? Die Micro-Consistency Checks sind Ihre Reflexionsmomente – explizite Pausen, um Ihr mentales Modell zu kalibrieren.

Hands-on: Mini-Projekt & Reflexion

4 Meilensteine begleiten unsere Reise:

- **MS1:** Rohdaten + Key-Value Layer
- **MS2:** Document Migration + Schema-Evolution
- **MS3:** Column Analytics + Performance-Benchmarks
- **MS4:** Relationales Redesign + Index-Strategien

Plus **Micro-Consistency Checks** nach jedem Block: „Was glaube ich jetzt – und was hat sich seit Block X verschoben?“

Am Ende verfügen Sie über ein begründbares Entscheidungsrepertoire: Sie können Anwendungsfälle auf Paradigmen abbilden, Trade-offs artikulieren und Risiken antizipieren. Das ist der Unterschied zwischen einem SQL-Kurs und einem Architektur-Kompass. Ein SQL-Kurs lehrt Syntax – `SELECT * FROM table`. Ein Architektur-Kompass lehrt Entscheidungsfindung: „Für Session-Storage brauche ich $O(1)O(1)$ Zugriff und TTL-Support – also Key-Value. Für Beziehungsanalyse brauche ich Traversierung – also Graph. Für Reportings brauche ich Aggregationen über Millionen Zeilen – also Column Store.“ Sie lernen nicht nur Tools, sondern wann und warum Sie sie einsetzen. Das ist polyglotte Denkweise: das richtige Werkzeug für den richtigen Job.

Ihr Kompass für datengetriebene Architektur

-  **Trade-off Verständnis:** Wann nutze ich was?
-  **Risiko-Antizipation:** Schema Drift, Lock Contention, Replikationsverzögerung
-  **Architektur-Entscheidungen:** Sessions vs. Metriken vs. Beziehungsanalyse
-  **Polyglot Thinking:** Das richtige Tool für den richtigen Job

Heute: Wir starten mit der Frage: „Was sind eigentlich Daten?“ 

Was sind Daten



Bevor wir uns in CSV-Dateien und JSON-Objekte stürzen, machen wir einen fundamentalen Schritt zurück: Was sind eigentlich Daten? Diese Frage ist nicht philosophisch gemeint, sondern praktisch.

Daten

Daten sind die rohe, uninterpretierte Ebene. Pixel, Bytes, Zeichen – alles ohne Kontext oder Bedeutung. In Datenbanken entspricht das den puren Feldwerten: „42“, „Schmidt“, „2023-10-03“. Schauen Sie sich dieses Bild an und beschreiben Sie nur das, was Sie sehen – ohne zu interpretieren, was es bedeuten könnte.

Daten können strukturiert oder unstrukturiert vorliegen, die zur Beschreibung von Objekten, Ereignissen oder Zuständen verwendet werden. Sie können in verschiedenen Formaten vorliegen.

Beispiel:

Was sehen Sie? Beschreiben Sie das folgende Bild möglichst detailliert:



Abb.: Plains-Krieger mit traditionellem Federkopfschmuck – erstellt mit ChatGPT

Sie haben vermutlich Federn gezählt, Farben benannt, Materialien identifiziert. Das sind die Rohdaten – messbar,zählbar, objektiv. Aber was bedeuten sie? Hier kommen Informationen ins Spiel.

Informationen

Informationen entstehen, wenn wir Daten Kontext und Bedeutung geben. Die sieben Adlerfedern sind nicht nur „sieben gelbe Objekte“ – sie sind kodierte Nachrichten mit spezifischer kultureller Bedeutung. In Datenbanken entspricht das der semantischen Ebene: Ein Feld „salary“ mit Wert „50000“ wird zur Information „Jahresgehalt: 50.000 Euro“. Schauen Sie, wie sich rohe Beobachtungen in bedeutungsvolle Nachrichten verwandeln:

Moment mal, was bedeutet das alles?

- **Adlerfeder aufrecht:** Tapferkeit im Kampf (eine Feder je getötetem Feind)
- **Falkenfeder schräg-links + rot:** Geschicklichkeit und Schnelligkeit (eine Feder je erfolgreichem Überraschungsangriff)
- **Eulenfeder hängend + schwarz:** Weisheit und Nachtsicht (eine Feder je überlebter Nachtschlacht)
- **Krähenfeder zentral:** Intelligenz und Anpassungsfähigkeit (eine Feder je erfolgreich gelöstem Problem)
- **Lederband mit Kerben:** Lebensjahre und Erfahrungen (eine Kerbe je Lebensjahr)
- **Pferdehaarknoten:** Mindestens ein erbeutetes Pferd

Wissen

Jetzt haben wir einzelne Bedeutungen, aber noch keine Gesamtsicht. Wissen entsteht, wenn wir Informationen systematisch verknüpfen und Muster erkennen. Die Anordnung und Anzahl der Federn, die Farben und Lederbandkerben sind kein Zufall – sie folgen einem kohärenten System zur Kodierung von Lebenserfahrungen. In Datenbanken entspricht das den Queries, die Beziehungen aufdecken: „Welche Kunden kaufen zusammen?“ oder „Welche Faktoren korrelieren mit Erfolg?“ Das Wissen liegt in den erkannten Zusammenhängen.

Die Anordnung und Anzahl der Federn, die Farben und die Lederbandkerben sind keine zufälligen Dekorationen, sondern kodieren spezifische Informationen über die Errungenschaften, Fähigkeiten und das Alter des Kriegers. Ein erfahrener Krieger mit vielen Federn und Kerben erzählt eine Geschichte von Mut, Geschicklichkeit, Weisheit und einem langen Leben voller Herausforderungen und Siege.

Aber Wissen allein reicht nicht. Weisheit entsteht, wenn wir aus Wissen handlungsrelevante Entscheidungen ableiten können. Hier wird es praktisch – und manchmal überlebenswichtig.

Weisheit

Weisheit ist angewandtes Wissen für Entscheidungen. Alle Informationen über den Krieger führen zu einer klaren Handlungsempfehlung: „Konflikt vermeiden!“ In Datenbanken entspricht das den Business Intelligence Systemen, die aus Mustern Aktionen ableiten: „Kunde X hat 80% Abwanderungsrisiko – sofort Retention-Maßnahmen einleiten.“ Weisheit macht Daten actionable.



THREAT ASSESSMENT: MAXIMUM



Combat Experience:	[]	10/10
Leadership Skills:	[]	8/10
Survival Instinct:	[]	10/10
Strategic Thinking:	[]	9/10
Recommendation:	AVOID CONFLICT	

„Würde ich diesem Typen das Pferd stehlen?“ → NEIN

„Würde ich ihn in meinem Team haben wollen?“ → DEFINITIV JA

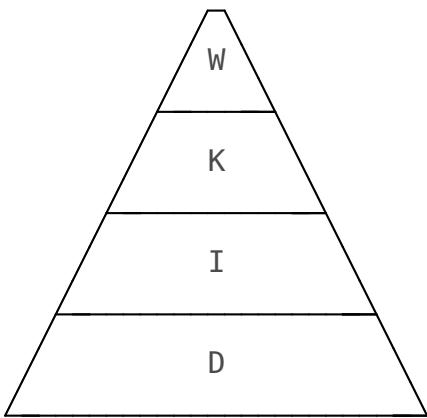
„Würde ich ihm widersprechen?“ → Nur sehr höflich

DIKW Pyramide

Jetzt konsolidieren wir unsere Erkenntnisse in der DIKW-Pyramide. Diese Hierarchie ist nicht nur akademische Theorie – sie ist das Fundament für jede Datenbankarchitektur. Jede Ebene stellt andere Anforderungen an Speicherung, Verarbeitung und Zugriff. Verstehen Sie DIKW, verstehen Sie, warum verschiedene Datenbank-Paradigmen existieren.

DIKW steht für **Data, Information, Knowledge, Wisdom** (Daten, Informationen, Wissen, Weisheit) und beschreibt eine Hierarchie der Verarbeitung und Bedeutung von Daten.

Die Pyramide zeigt uns vier Stufen der Wertschöpfung. Unten haben wir das Rohmaterial – Daten. Jede Stufe nach oben wird wertvoller, aber auch komplexer zu verarbeiten. In Datenbanken entspricht jede Ebene verschiedenen Systemanforderungen.



4. Wisdom (Weisheit)
3. Knowledge (Wissen)
2. Information (Informationen)
1. Data (Daten)

Lassen Sie uns jede Ebene mit ihren Datenbank-Entsprechungen verstehen. Die Basis: Daten sind die rohen Feldwerte. Informationen fügen Semantik hinzu. Wissen erkennt Muster durch Queries. Und Weisheit leitet Aktionen ab – das ist Business Intelligence.

Die vier Ebenen in Datenbank-Kontext:

- **Daten:** Rohe Feldwerte ohne Kontext (`"42"`, `"Schmidt"`, `"2023-10-03"`)
- **Information:** Semantische Bedeutung (`"Alter: 42 Jahre"`, `"Nachname: Schmidt"`)
- **Wissen:** Muster durch Queries (`"Kunden über 40 kaufen häufiger Produkt X"`)
- **Weisheit:**
Handlungsempfehlungen (`"Kunde Schmidt → Produkt X empfehlen"`)

Unser Indianer-Beispiel durchläuft genau diese Stufen: Rohe Pixel werden zu kulturellen Codes, diese zu systematischem Verständnis, und schließlich zu der weisen Entscheidung: „Besser nicht ärgern!“ Jede Datenbank-Anwendung macht dieselbe Reise.

Datenorganisation – Die Ewige Suche nach Ordnung

Hier ist eine unbequeme Wahrheit: Jede Datenbank löst ein Problem, das Menschen jahrhundertelang von Hand gemacht haben. Und oft haben sie es besser gemacht als wir heute glauben.

1970 prägte Edgar Codd, der Erfinder der relationalen Datenbanken, den folgenden Satz: „The relational model provides a means of describing data with its natural structure only“

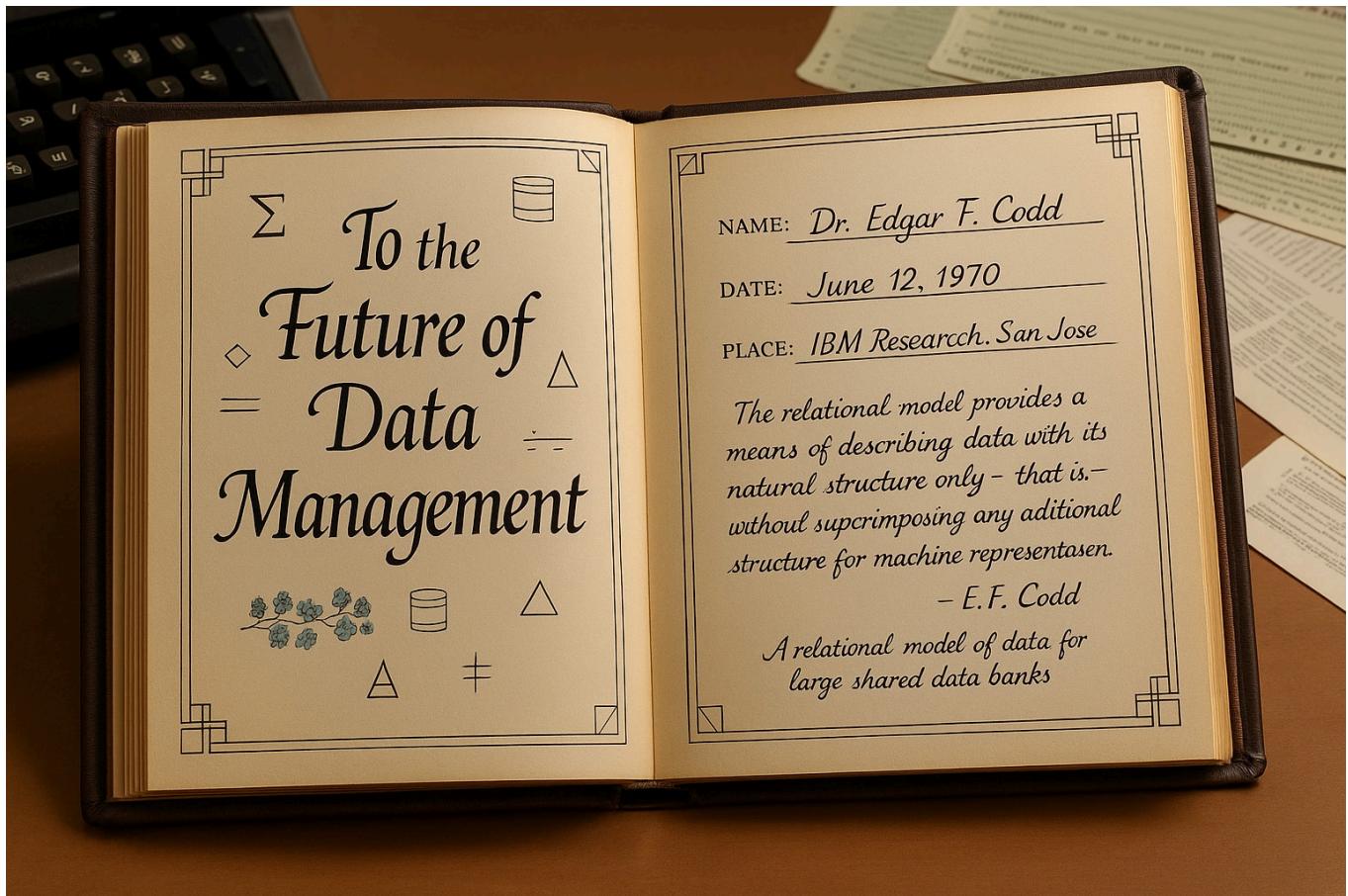


Abb.: E. F. Codds Freundschaftsalbum-Eintrag 1970 – Relationale Datenbank – erstellt mit ChatGPT

Die Ironie? Seine „revolutionäre“ Idee war ein Rückschritt zu dem, was Bibliothekare seit 100 Jahren mit Karteikarten machten!

Und Ja, ein Poesiealbum ist tatsächlich eine Art Datenbank – mit Einträgen, Attributen, einem impliziten Schema und Integritätsregeln. Es ist theoretisch und praktisch ein spezialisiertes und handgefertigtes, physisches Datenbanksystem, das Menschen seit Generationen nutzen.

Bevor wir uns in NoSQL, NewSQL und anderen Buzzwords verlieren, schauen wir zurück: Was haben Menschen früher richtig gemacht? Welche Probleme haben sie elegant gelöst? Und wo sind sie gescheitert – sodass wir Maschinen brauchten?

Historische Beispiele für Datenorganisation

Hier beginnt eine faszinierende Entdeckungsreise: Jede moderne Datenbank löst Probleme, die Menschen jahrhundertelang von Hand gemeistert haben. Und erstaunlicherweise haben unsere Vorfahren oft elegantere Lösungen gefunden, als wir heute glauben. Lassen Sie uns die DNA moderner Datenspeicherung in historischen Systemen aufspüren.

Unsere Zeitreise durch 5000 Jahre Datenorganisation:

Jede „revolutionäre“ Datentechnologie hat historische Wurzeln. Wir lernen nicht nur, **was** funktioniert – sondern **warum** es schon immer funktionierte.

Beginnen wir bei den Anfängen: Die alten Ägypter und Sumerer hatten bereits ein kritisches Problem erkannt – wie dokumentiert man Mengen und Verteilungen so, dass Betrug schwierig wird und Nachvollziehbarkeit gewährleistet ist? Ihre Tontafeln und Papyri waren die ersten „Audit Trails“ der Menschheitsgeschichte.

1. Frühe Verwaltungslisten (3000 v.Chr. - 500 n.Chr.) 📜



Precuneiform table -AO 29562 (Louvre) — Photo: Pouppy / Zunkir, Wikimedia Commons, CC BY-SA 3.0

Diese frühen Systeme verfolgten ein klares Ziel: verlässliche Mengennachweise und Verteilungskontrolle. Die Herausforderungen? Doppelte Einträge, mehrdeutige Interpretationen und fehlende Standardisierung untergruben das Vertrauen und machten Abgleiche extrem aufwendig.

Problem gelöst: Dauerhafte Persistenz, Inkonsistenz-Prävention



GETREIDESPEICHER BABYLON (ca. 2500 v.Chr.)

DATUM: 3. MOND NISANNU
EMPFÄNGER: HAMMURABI-SOHN

MENGE: | | | | | | | | (10 Maß Gerste)
KONTROLLE: □ □ □ (Doppelt eingetragen)

Moderne DNA: Append-Only Logs, Audit Trails

Springen wir ins Mittelalter: Luca Pacioli revolutionierte 1494 das Rechnungswesen mit einem eleganten Trick – jede Transaktion wird doppelt erfasst. Soll und Haben müssen sich ausgleichen, sonst stimmt etwas nicht. Das ist die Ur-Idee der Transaktions-Integrität!

2. Doppelte Buchführung (~1494)



Portrait of Luca Pacioli (1445–1517) with a student, attrib. Jacopo de' Barbari — Wikimedia Commons, Public Domain.

Paciolis Genialität lag in der Invariante: Jede Buchung erzeugt einen ausgleichenden Gegeneintrag. Fehler zeigen sich sofort als Unbalancen. Dieses Prinzip ist das Fundament für moderne ACID-Transaktionen – Atomizität durch systematische Dualität.

Im Beispiel sehen wir, wie jede Transaktion zwei Seiten hat – Soll und Haben. Am Ende muss die Summe beider Seiten übereinstimmen. Im ersten Eintrag wird Geld in die Kasse gelegt (Soll), im zweiten werden Waren gekauft (Haben). Danach werden gegen einen Kredit Waren eingekauft (Soll). Im letzten Eintrag wird ein Gewinn (Soll) aus der Kasse entnommen (Haben). Am Ende stimmen die Summen überein – die Bücher sind ausgeglichen.

Problem gelöst: Transaktions-Integrität, Physische Skalierung



HANDELSBUCH VENEDIG (1494)

SOLL		HABEN	
Kassa	100 €	Waren	-100 €
Waren	200 €	Kredit	-200 €
Gewinn	50 €	Kassa	-50 €
SUMME:	350 €	SUMME:	350 €

✓

Moderne DNA: ACID-Transaktionen, Two-Phase Commit

Die Seefahrt brachte eine weitere Innovation: das Logbuch. Jeder Eintrag ist zeitgestempelt, nichts wird gelöscht, alles wird nacheinander aufgeschrieben. Bei Schiffsunglücken oder Rechtsstreitigkeiten war die lückenlose Chronologie überlebenswichtig. Das ist das Ur-Prinzip des Write-Ahead Logs!

3. Schiffslogbücher & Navigationsjournale

LOG OF JENNY 2							
From	LORD HOWE	to	RAPA ITI	Date	TUES 16/4 1963		
HOUR	LOG	DISTANCE MADE GOOD	COURSE	WIND DIRECTION AND FORCE	BAROMETER	LEEWAY	REMARKS
A.M. 9 1	299.3	5.0	156	SW F5-6	29.9	NIL	HEAVY SEAS But don't stay
2	305.	5.7	156	SW F5-6	29.9	NIL	On set as we are running to
3 3	308	3.0	156	SW 5-6	29.9	NIL	WHAT corresponds with your
4	311	3.0	156	SW 5-6	29.9	NIL	20 knots of breaking wave front 100
5	319.5	8.5	156	SW 5-6	29.9	NIL	Very uncomfortable down below
6	330	8.5	156	SW 8-	29.9	NIL	decks.
7	337.4	9.4	156	SW 7	29.95	NIL	SEAS HEAVY HUGE SWELLS
8	344.8	9.4	156	SW 7	29.95	NIL	
9	350.4	6.4	156	SW 7	29.95	NIL	30 FT SWELLS CREST TO PEAK
10	359.4	6.4	156	SW 7	29.95	NIL	
11	364.4	7.0	156	SW 7	29.95	NIL	
N. 12	371.2	6.8	156	SW 7	29.95	NIL	
P.M. 1	376.9	5.9	156	SW 5-6	29.9	NIL	SEAS TIEFADING
2	382.7	5.7	156	SW 4-5	29.9	NIL	" "
3	387.7	5.0	156	S 5-6	29.9	NIL	
4	392.7	5.0	156	SE 5-6	29.9	NIL	
5	397.7	5.0	156	SE 5-6	29.95	NIL	
6	401.1	6.4	156	SE 5-6	30.0	NIL	VERY NEARLY TREACHEROUS
7	410	8.0	156	SE 5-6	30.1	NIL	HER MAXIMUM HULL SPEED
8	419	9.0	156	SE 5-6	30.1	NIL	THIS IS HER 33EST POINT OF
9	428	9.0	156	E/E 5-6	30.1	NIL	SAILING WITH THE WIND
10	437.9	9.9	156	SE 6	30.1	NIL	ABAIT OF 150M 9 NOT 10
11	445.2	4.8	156	SE 6	30.1	NIL	THE HOUR
12	452.6	7.3	156	SE 6	30.1	NIL	Moderate seas

Autor: Belleami / Cythera. Public Domain. Wikimedia Commons

Schiffslogbücher etablierten das Append-Only Prinzip: Kurs, Wetter und Ereignisse werden chronologisch festgehalten – niemals überschrieben, niemals gelöscht. Die Herausforderung? Uneinheitliche Formate und schwierige Querverweise machten Analysen mühsam.

Problem gelöst: ✓ Append-Only Durability, ✗ Strukturierte Abfragen



LOGBUCH HMS BEAGLE (1831-1836)

Tag 847: 15° S, 47° W | Wind: S 0 4 bft
Darwin sammelt Finken | Wasser: 300 Ltr

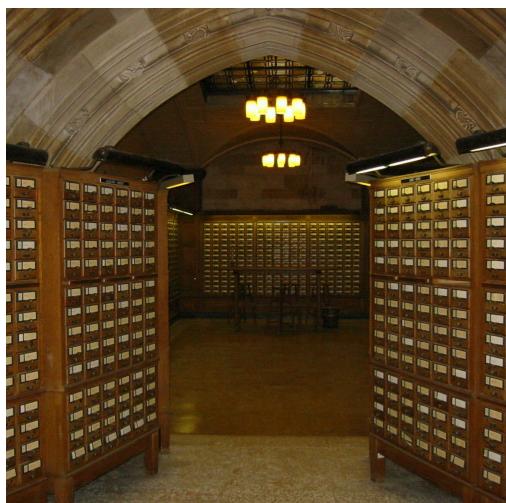
Tag 848: 16° S, 48° W | Wind: 0 2 bft
Sturm voraus sichtbar | Segel gerefft

Tag 849: POSITION UNBEKANNT | Kompass
defekt | Darwin seekrank | NOTLAGE!

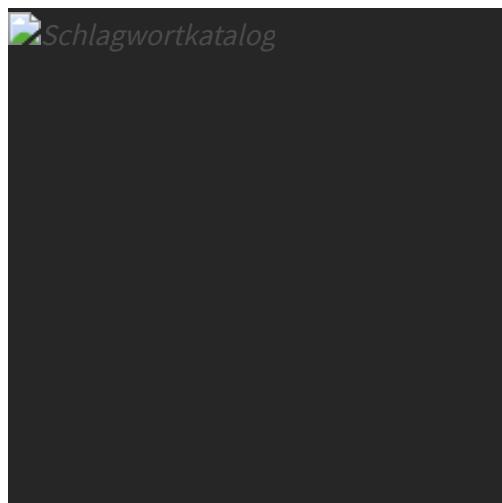
Moderne DNA: Write-Ahead Logs, Event Sourcing

Ende des 19. Jahrhunderts perfektionierten Bibliothekare ein System, das heute noch beeindruckt: Katalogkarten. Für jedes Buch mehrere Zugriffspfade – nach Autor, Titel, Thema. Das ist die Erfindung der Sekundärindizes! Millionen von Büchern, gefunden in Sekunden.

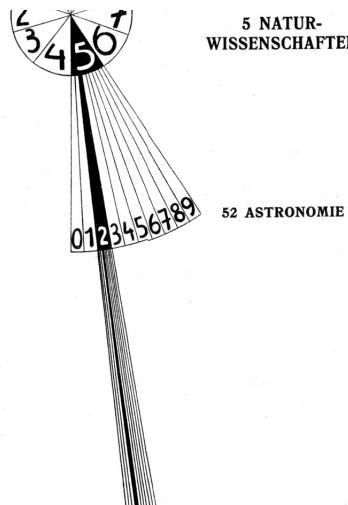
4. Bibliotheks-Katalogkarten (Dewey Decimal, 1876)



Kartei-System der Yale University Library



Schlagwortkatalog — Foto:
Bernhard/Wikimedia Commons, Lizenz: CC
BY-SA 4.0 (commons.wikimedia.org)



Dewey Decimal Classification System -
Astronomie

Melvil Dewey und seine Zeitgenossen lösten das Indexierungs-Problem elegant: Mehrere Zugriffspfade auf dieselben Daten, systematische Kategorisierung, und $O(\log n)$ Suchzeit durch alphabetische Ordnung. Probleme entstanden durch physische Fragmentierung und Redundanz-Management – aber das Grundprinzip war brillant.

Problem gelöst: Sekundäre Zugriffspfade, Konsistenz bei Updates



DEWEY DECIMAL SYSTEM – INDEX DESIGN

HAUPTKATALOG (nach Standort)

510.123 → Mathematik: Algebra Bd.3

AUTORENKATALOG (nach Nachname)

Einstein → [510.543, 523.877, —]

SCHLAGWORTKATALOG (nach Thema)

Relativität → [523.877, 510.543]

PROBLEM: 3x Redundanz, manueller Sync!

Moderne DNA: B-Tree Indizes, Multi-Column Indexing

1890 revolutionierte Herman Hollerith die Datenverarbeitung: Standardisierte Lochkarten für den US-Zensus. Erstmals konnten Millionen von Datensätzen maschinell ausgewertet werden. Der Preis? Ein starres Schema – jede Änderung bedeutete neue Hardware-Konfiguration.

5. Hollerith-Lochkarten (US Census 1890)

Punchkarte für Herman Holleriths elektrische Sortier- und Tabuliermaschine, ca. 1895 — Public Domain (Library of Congress).
Wikimedia Commons.

Holleriths Innovation war die physische Standardisierung: Jede Karte hatte exakt dieselbe Struktur, jede Position eine definierte Bedeutung. Das ermöglichte erstmals automatisierte Statistiken über Millionen Menschen. Aber Schema-Änderungen bedeuteten Hardware-Umbau – Flexibilität war unmöglich.

Problem gelöst:  Maschinelle Auswertung,  Schema-Flexibilität



HOLLERITH CENSUS CARD (1890)

Col 1-2: AGE	[●●]	= 42 Jahre
Col 3: SEX	[●]	= Male
Col 4-5: STATE	[●●]	= New York
Col 6: MARRIED	[●]	= Yes
Col 7-8: OCCUP	[●●]	= Farmer
Col 9: LITERATE	[●]	= Can Read
Col 10: CITIZEN	[●]	= Native Born

80 Spalten, fixes Format, Maschine liest
500 Karten/Minute! (vs. Jahre manuell)

Moderne DNA: Schema-Enforcer, Column-Based Storage

Und damit schließt sich der Kreis zu unserem Eingangsbild: Edgar Codd's „revolutionäre“ relationale Theorie war im Grunde eine Rückkehr zu den Katalogkarten-Prinzipien – aber mit mathematischer Präzision und maschineller Flexibilität. Die Ironie der Informatikgeschichte!

6. Der Kreis schließt sich: Codd 1970 → Heute

Schauen Sie zurück auf unsere Zeitreise: Jedes „neue“ Datenbankkonzept hat historische Wurzeln. Append-Only Logs? Schiffstagebücher. ACID-Transaktionen? Doppelte Buchführung. Sekundärindizes? Katalogkarten. Die Innovation liegt nicht in der Erfindung neuer Prinzipien – sondern in deren maschineller Perfektionierung.

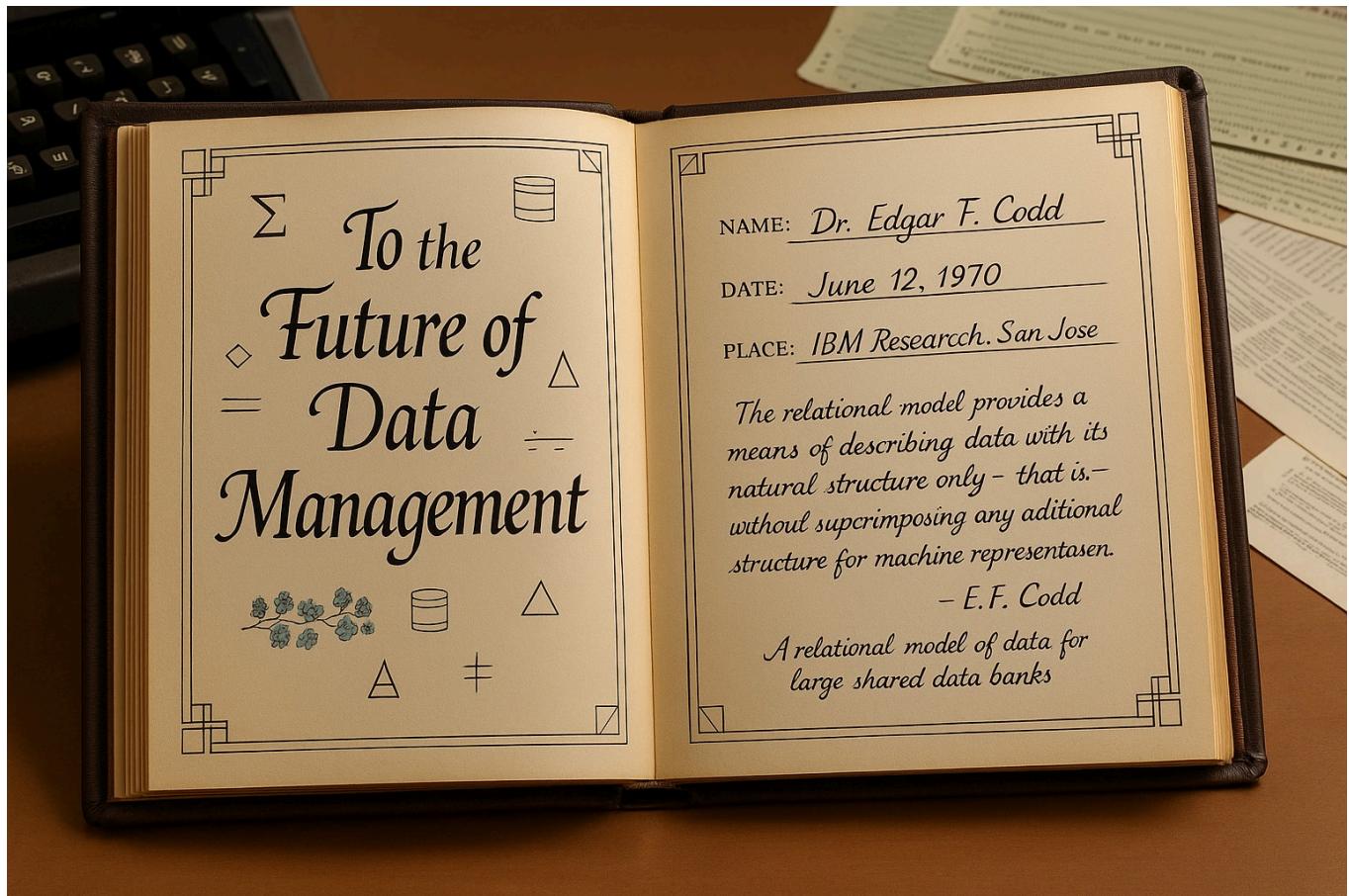
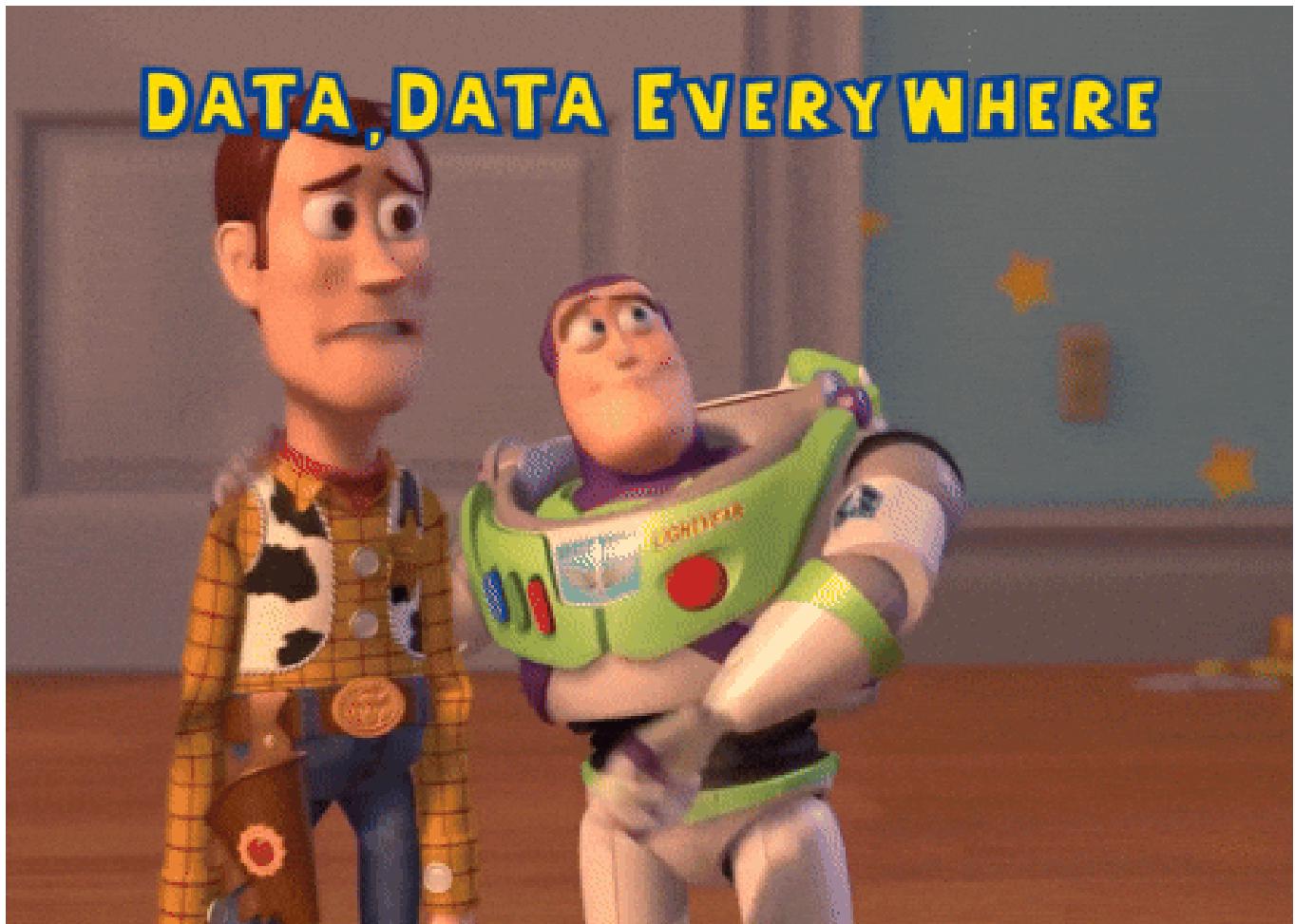


Abb.: E. F. Codds Freundschaftsalbum-Eintrag 1970 – Relationale Datenbank – erstellt mit ChatGPT

Datenformate im Vergleich: CSV, JSON, XML & Co.

Wir haben die historischen Grundlagen gelegt – jetzt wird es praktisch! In der nächsten Session nehmen wir uns reale Datenformate vor und sehen, wo sie uns im Stich lassen. CSV, JSON, XML – jedes Format hat seine Berechtigung, aber auch seine Tücken.



CSV - Das trojanische Pferd der Datenformate 🐾

CSV - Comma Separated Values - ist das Schweizer Taschenmesser der Datenwelt. Jeder kennt es, jeder nutzt es, und jeder unterschätzt seine Tücken. Als praxisorientierter Architekt sage ich Ihnen: CSV ist gleichzeitig das nützlichste und gefährlichste Format, das Sie je verwenden werden.

Was ist CSV eigentlich?

CSV steht für **Comma Separated Values** - eine scheinbar simple Textdatei, in der Datensätze zeilenweise und Felder durch Kommas getrennt gespeichert werden.

Die Ironie: Ein „Standard“ ohne echte Standardisierung - RFC 4180 kam erst 2005, nachdem CSV bereits 20 Jahre wild gewachsen war!

Schauen wir uns die Anatomie einer CSV-Datei an. Auf den ersten Blick wirkt es harmlos - Zeilen, Kommas, fertig. Aber der Teufel steckt im Detail: Was passiert, wenn ein Feld selbst ein Komma enthält? Wie codiere ich Anführungszeichen? Welche Zeichen sind Zeilenenden?

Die Anatomie einer CSV-Datei

Problem with the value at json.ok: {} Expecting a BOOL

Grundregeln (die niemand einheitlich befolgt):

- Erste Zeile = Header (meist, aber nicht immer)
- Komma = Feldtrenner (außer es ist Semikolon, Tab, oder Pipe)
- Anführungszeichen für Felder mit Sonderzeichen (wenn überhaupt)
- Zeilenende = Datensatz-Ende (CR, LF, oder CRLF?)

Hier beginnt das CSV-Drama! Jeder implementiert es anders. Excel nutzt regional unterschiedliche Trennzeichen - in Deutschland Semikolon statt Komma, weil Komma als Dezimaltrennzeichen dient. Französische Systeme nutzen andere Kodierungen. Und dann gibt es noch die Escape-Hölle: Wie schreibt man Anführungszeichen in ein angeführtes Feld?

⚠ Die CSV-Hölle: Wilde Varianten

INTERNATIONALE CSV-VERWIRRUNG			
US-Style,	"Smith, John",	42,50000	
German-Style,	"Smith, John";	42;50000	
French-Style,	"Smith, John";	42;50000	
Unix-Style,	Smith\, John,	42,50000	
Excel-Export,	"Smith, John",	42,50000	
Database-Dump,	'Smith, John',	42,50000	

Die vier Hauptprobleme:

1. **Trennzeichen-Chaos:** [,] vs [;] vs [\t] vs [|]
2. **Escape-Hölle:** Wie kodiere ich [" in "Feld"] ?
3. **Encoding-Wirrwarr:** UTF-8 vs Latin-1 vs Windows-1252
4. **Implizite Typen:** Ist "42" Text oder Zahl?

Lassen Sie uns praktisch werden! Hier ist ein echter CSV-Alptraum aus der Praxis. Schauen Sie genau hin - können Sie alle Probleme erkennen? Inkonsistente Anführungszeichen, mixed Encodings, verschiedene Datumsformate, und sogar eingebettete Zeilenwechsel. Das ist leider Realität, nicht Übertreibung.

⚠ CSV-Alptraum aus der Praxis (Titanic-Export vom Legacy-System)

PassengerId, Survived, Pclass, Name, Sex, Age, SibSp, Parch, Ticket, Fare, Cabin, Embarked
1,0,3,"Braund, Mr. Owen Harris",male,1912-04-15,1,0,A/5 21171,7.25,,S
2,1,1,Cumings, Mrs. John Bradley (Florence Briggs Thayer),female,38,1,0,PC ,71.2833,C85,Cherbourg
3,1,3,"Heikkinen, Miss. Laina",female,26.0,0,0,STON/O2. 3101282,7.925 ,Southampton
"4",1,1,"Futrelle, Mrs. Jacques Heath (Lily May Peel)
Notiz: Überlebte in Rettungsboot 9",female,35,1,0,113803,53,1€,C123,S
5,0,3,"Allen, Mr. William Henry",mâle,35,0,0,373450,8.05,NULL,S

Probleme identifiziert (alle in 5 Zeilen!):

1. **Inkonsistente Datumsformate:** `1912-04-15` (ISO) vs. `38` (Alter) vs. `26.0` (Float) – Age-Spalte gemischt!
2. **Inkonsistente Quotes:** Zeile 2 fehlt Opening-Quote bei Name → Parser-Chaos
3. **Mixed Encoding:** `mâle` (French UTF-8) statt `male` → Latin-1/UTF-8 Konflikt
4. **Multiline Fields:** Zeile 4 hat Zeilenwechsel in Name-Feld → „Notiz: Überlebte...“
5. **Quoted Numbers:** `"4"` als PassengerId → String statt Integer
6. **Type Confusion:** `7.25` vs `53,1€` vs `53` – verschiedene Währungs-/Dezimalnotationen
7. **Mixed Value Semantics:** (leer) vs `NULL` vs fehlende Spalte – drei Arten von „missing“
8. **Inconsistent Categories:** `S` vs `Southampton` vs `Cherbourg` – mal Code, mal Volltext

Real-World Impact: Ein einziger falsch escapeter Zeilenwechsel kann eine komplette Datenimport-Pipeline zum Absturz bringen!

Warum wird CSV trotz all dieser Probleme so viel verwendet? Weil es funktioniert - meistens. Es ist das kleinste gemeinsame Vielfache aller Systeme. Jede Programmiersprache kann es lesen, jede Datenbank kann es importieren, jeder Mensch kann es verstehen. Es ist der Duct-Tape der Datenwelt.

Warum CSV trotzdem überlebt

Die Stärken:

- **Universal lesbar:** Von Excel bis Python, von MySQL bis Notepad
- **Menschenlesbar:** Man sieht sofort, was drin steht
- **Kompakt:** Wenig Overhead, hohe Datendichte
- **Stream-fähig:** Kann zeilenweise verarbeitet werden
- **Git-freundlich:** Textbasiert, diff-bar, mergeable

Die Schwächen:

- **Schemalos:** Keine Typdefinitionen, keine Validierung
- **Fehleranfällig:** Silent Failures bei Parsing-Problemen
- **Begrenzt:** Keine Verschachtelung, keine Metadaten
- **Inkonsistent:** Jeder interpretiert den „Standard“ anders

Hier ist Ihr Praxis-Guide für den Umgang mit CSV. Diese Regeln haben mir in 15 Jahren Datenarchitektur viel Ärger erspart. Definieren Sie IMMER das Schema explizit, nutzen Sie UTF-8 BOM für Excel-Kompatibilität, und testen Sie mit echten Daten - nicht nur mit Ihren sauberen Testdatensätzen.

CSV-Survival-Guide für Profis

Beim CSV-Import (Sie kriegen Daten rein):

```

1 import pyodide.http
2 import pandas as pd
3 from io import StringIO
4
5 v async def fetch_data(url):
6     response = await pyodide.http.pyfetch(url) # Daten abrufen
7     csv_data = await response.string() # Inhalt als String laden
8     df = pd.read_csv(StringIO(csv_data)) # In DataFrame umwandeln
9     return df
10
11 # Funktion direkt aufrufen mit `await` (NICHT `asyncio.run()`)
12 url = "https://raw.githubusercontent.com/andre-dietrich/Datenbankensys"
13 Titanic = await fetch_data(url)
14
15 # Ausgabe
16 Titanic

```

```

1 import matplotlib.pyplot as plt
2
3 # Absolute Häufigkeit der Überlebenden und Nicht-Überlebenden berechne
4 absolute_counts = Titanic.groupby(["Pclass", "Sex"])["Survived"].value_
5
6 # Visualisierung der absoluten Häufigkeiten
7 absolute_counts.plot(kind="bar", stacked=True, figsize=(10,6), edgecolor="black")
8 plt.title("Absolute Häufigkeit der Überlebenden nach Passagierklasse u"
9 plt.xlabel("Passagierklasse und Geschlecht")
10 plt.ylabel("Anzahl der Passagiere")
11 plt.xticks(rotation=0)
12 plt.legend(["Nicht Überlebt", "Überlebt"], title="Status")
13 plt.grid(axis="y", linestyle="--", alpha=0.7)
14 plt.show()
15 plt

```

```

1 library(ggplot2)
2 library(dplyr)
3
4 # CSV-Datei einlesen
5 df <- read.csv("https://raw.githubusercontent.com/andre-dietrich
6                 /Datenbankensysteme-Vorlesung/refs/heads/main/assets/dat/titanic.cs
7
8 # Alter bereinigen (NA-Werte entfernen)
9 df <- df %>% filter(!is.na(Age))
10
11 # Überlebenswahrscheinlichkeit nach Geschlecht und Alter (inkl. Männe

```

```

11 women_children_men <- d %>%
12   mutate(Category = case_when(
13     Sex == "female" & Age < 18 ~ "Female Child",
14     Sex == "female" & Age >= 18 ~ "Female Adult",
15     Sex == "male" & Age < 18 ~ "Male Child",
16     Sex == "male" & Age >= 18 ~ "Male Adult"
17   )) %>%
18   group_by(Category) %>%
19   summarise(SurvivalRate = mean(Survived), .groups = 'drop')
20
21 # PNG-Datei für Analyse speichern
22 png("women_children_men_survival.png", width = 800, height = 400)
23
24 ggplot(women_children_men, aes(x = Category, y = SurvivalRate, fill =
25   Category)) +
26   geom_bar(stat = "identity", position = "dodge") +
27   ggttitle("Überlebenswahrscheinlichkeit von Frauen, Männern und Kinde
28   xlabel("Kategorie") +
29   ylab("Überlebensrate") +
30   scale_fill_manual(values = c("blue", "red", "green", "purple"), nam
31   "Kategorie") +
32   theme_minimal()

```

```
Attaching package: 'dplyr'
```

```
The following objects are masked from 'package:stats':
```

```
  filter, lag
```

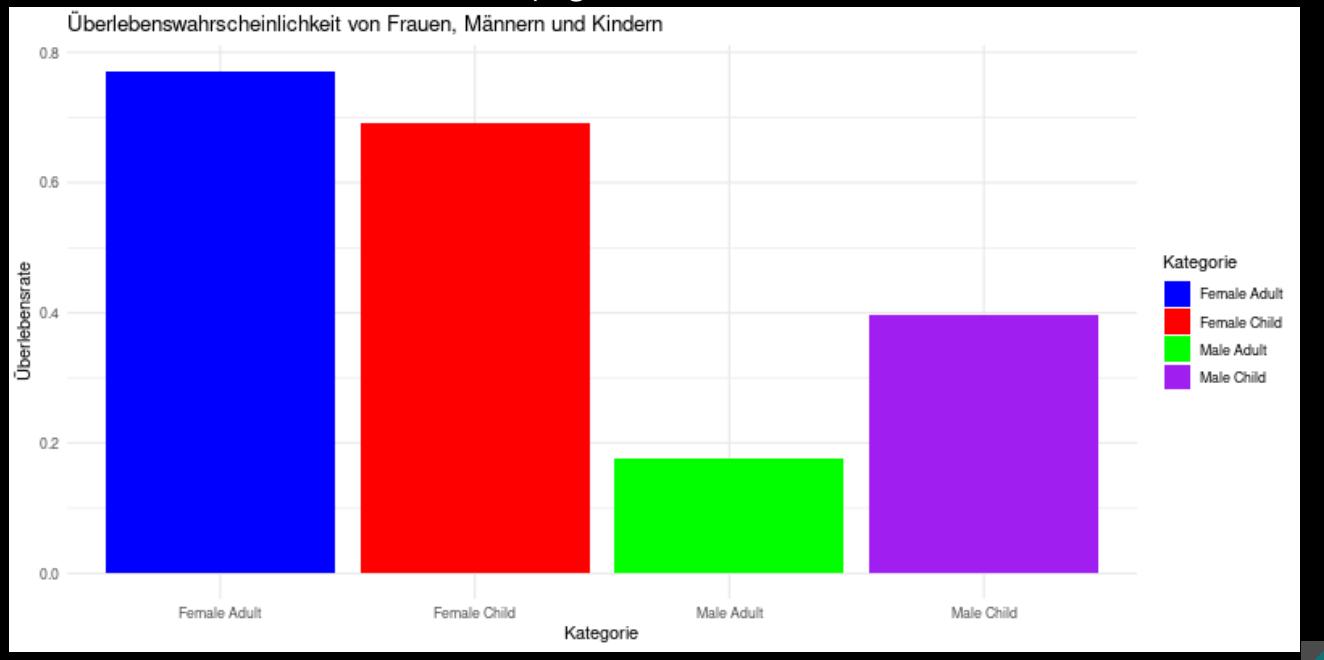
```
The following objects are masked from 'package:base':
```

```
  intersect, setdiff, setequal, union
```

```
null device
```

```
  1
```

women_children_men_survival.png



Zum Abschluss: CSV ist wie ein Schraubendreher - für manche Aufgaben perfekt, für andere völlig ungeeignet. Nutzen Sie es für Datenexporte, ETL-Zwischenschritte und schnelle Analysen. Aber bauen Sie nie eine Anwendung darauf auf. In der nächsten Session sehen wir, wie JSON versucht, CSVs Schwächen zu beheben.

🎯 Wann CSV verwenden - Wann nicht

✓ CSV ist perfekt für:

- **Datenexport** aus Datenbanken für Analysen
- **ETL-Pipelines** als Zwischenformat
- **Reporting** für Business-User (Excel-Import)
- **Data Science** für schnelle explorative Analysen
- **Logs** mit strukturierten Events

CSV ist schlecht für:

- **Produktive Datenhaltung** (keine Integrität)
- **APIs** (keine Typisierung, kein Schema)
- **Hierarchische Daten** (keine Verschachtelung)
- **Multi-User Szenarien** (Concurrency-Probleme)
- **Komplexe Queries** (keine JOIN-Unterstützung)

Professor Freinets Regel: CSV für Transport, nie für Storage!

XML - Als Ordnung zur Bürokratie wurde

XML - Extensible Markup Language - ist das formale Gegenstück zu CSV. Wo CSV zu locker ist, schoss XML über das Ziel hinaus. Als praxisorientierter Architekt sage ich Ihnen: XML ist wie ein übervorsichtiger Anwalt - jedes Detail wird geprüft, jede Regel befolgt, aber am Ende dauert alles dreimal so lange.

Was ist XML eigentlich?

XML steht für **Extensible Markup Language** - ein textbasiertes Format für **selbstbeschreibende, hierarchische Dokumente** mit strenger Validierung.

Die Mission: Ordnung schaffen wo CSV Chaos hinterließ - mit formalen Schemas, Namespaces und Validierung.

Das Resultat: Der bürokratische Overkill - mehr Zeilen Schema als Daten, mehr Parser-Overhead als Nutzen.

Die Anatomie einer XML-Datei

Schauen wir uns die Anatomie von XML an. Auf den ersten Blick sieht es aus wie HTML - Tags, Attribute, Hierarchie. Aber der Unterschied ist fundamental: HTML ist nachsichtig („best effort parsing“), XML ist streng („well-formed or fail“). Ein einziges fehlendes Closing-Tag und der Parser verweigert die Arbeit.

```
<?xml version="1.0" encoding="UTF-8"?>
<library>
  <book isbn="978-0-12-110362-7">
```



```

<book isbn="978-0-13-110502-1">
  <title>The C Programming Language</title>
  <authors>
    <author>Brian Kernighan</author>
    <author>Dennis Ritchie</author>
  </authors>
  <year>1978</year>
  <price currency="USD">42.50</price>
</book>
<book isbn="978-0-13-468599-1">
  <title>Structure and Interpretation of Computer Programs</title>
  <authors>
    <author>Harold Abelson</author>
    <author>Gerald Jay Sussman</author>
  </authors>
  <year>1985</year>
  <price currency="USD">65.00</price>
</book>
</library>

```

Grundelemente:

- **Prolog:** `<?xml version="1.0" encoding="UTF-8"?>` (optional, aber empfohlen)
- **Root-Element:** Genau ein umschließendes Element (hier `<library>`)
- **Tags:** Öffnend und schließend, case-sensitive: `<book>...</book>`
- **Attribute:** Key-Value Paare in öffnenden Tags: `isbn="..."`
- **Hierarchie:** Beliebig tiefe Verschachtelung möglich

Well-formed vs. Valid - Der Unterschied

XML unterscheidet zwei Qualitätsstufen: „well-formed“ und „valid“. Well-formed bedeutet syntaktisch korrekt - alle Tags geschlossen, keine ungültigen Zeichen, ein Root-Element. Valid bedeutet zusätzlich: entspricht einem Schema. Und hier beginnt die XML-Hölle.

Well-formed = Syntaktisch korrekt

```

<!-- ✓ Well-formed -->
<person>
  <name>John</name>
  <age>42</age>
</person>

<!-- ✗ NOT well-formed: Missing closing tag -->
<person>
  <name>John</name>
  <age>42
</person>

<!-- ✗ NOT well-formed: Case mismatch -->

```

```
<Person>
  <name>John</name>
</person>
```

Valid = Entspricht einem Schema (DTD, XSD, RELAX NG)

```
<!-- Schema sagt: "age" muss Integer sein -->
<!-- ✓ Valid -->
<person>
  <name>John</name>
  <age>42</age>
</person>

<!-- ✗ NOT valid: age ist kein Integer -->
<person>
  <name>John</name>
  <age>forty-two</age>
</person>
```



Das Schema-Ökosystem - Drei Wege zur Validierung

Jetzt kommen wir zum XML-Ökosystem der Schema-Sprachen. Es gibt drei Hauptansätze: DTD aus den 90ern - einfach aber limitiert. XSD, der W3C-Standard - mächtig aber monströs verbose. Und RELAX NG - elegant aber kaum verbreitet. Das Problem? Für jede dieser Sprachen brauchen Sie ein eigenes Buch.

XML SCHEMA EVOLUTION

- | | |
|-----------------|----------------------|
| DTD (1990s) | – Simple, limited |
| XSD (2001) | – Powerful, VERBOSE |
| RELAX NG (2003) | – Elegant, unpopular |

DTD (Document Type Definition):

```
<!DOCTYPE library [
  <!ELEMENT library (book+)>
  <!ELEMENT book (title, authors, year, price)>
  <!ELEMENT title (#PCDATA)>
  <!ELEMENT authors (author+)>
  <!ELEMENT author (#PCDATA)>
  <!ELEMENT year (#PCDATA)>
  <!ELEMENT price (#PCDATA)>
  <!ATTLIST book isbn CDATA #REQUIRED>
  <!ATTLIST price currency CDATA #IMPLIED>
]>
```



XSD (XML Schema Definition):

```
<xsschema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xselement name="library">
    <xsccomplexType>
      <xsssequence>
        <xselement name="book" maxOccurs="unbounded">
          <xsccomplexType>
            <xsssequence>
              <xselement name="title" type="xs:string"></xselement>
              <xselement name="authors">
                <xsccomplexType>
                  <xsssequence>
                    <xselement name="author" type="xs:string" maxOccurs
                      ="unbounded"></xselement>
                  </xsssequence>
                </xsccomplexType>
              </xselement>
              <xselement name="year" type="xs:integer"></xselement>
              <xselement name="price">
                <xsccomplexType>
                  <xssimpleContent>
                    <xsextension base="xs:decimal">
                      <xssattribute name="currency" type="xs:string"></xs
                        :attribute>
                    </xsextension>
                  </xssimpleContent>
                </xsccomplexType>
              </xselement>
            </xsssequence>
            <xssattribute name="isbn" type="xs:string" use="required"></xs
              :attribute>
          </xsccomplexType>
        </xselement>
      </xsssequence>
    </xsccomplexType>
  </xselement>
</xsschema>
```

Das Paradox: 40 Zeilen Schema für 12 Zeilen Daten! 🤯

Lassen Sie uns das Kernproblem von XML visualisieren: Verbosity. Jede Information wird mehrfach kodiert - im öffnenden Tag, im schließenden Tag, und eventuell nochmal im Schema. Das ist wie ein Brief, bei dem Sie auf jedem Umschlag dreimal Ihre Adresse schreiben müssen.

👀 Der Verbosity-Alptraum - XML vs. Alternativen

 GLEICHE PERSON – VERSCHIEDENE FORMATE

CSV (30 Bytes):	
John Smith,42,Engineer,New York	
JSON (85 Bytes):	
{	
"name": "John Smith",	
"age": 42,	
"job": "Engineer",	
"city": "New York"	
}	
XML (187 Bytes):	
<?xml version="1.0"?>	
<person>	
<name>John Smith</name>	
<age>42</age>	
<job>Engineer</job>	
<city>New York</city>	
</person>	
Overhead: CSV 1x JSON 2.8x XML 6.2x	

Real-World Impact:

- **Bandbreite:** 6x mehr Datenvolumen als CSV
- **Parsing:** 10-50x langsamer als JSON (je nach Parser)
- **Lesbarkeit:** Menschen ertrinken in Closing-Tags
- **Wartung:** Schema-Änderungen sind Albträume

Die Ironie: XML sollte „human-readable“ sein, aber niemand liest gerne 187 Bytes statt 30.

🔍 XPath & XQuery - Die mächtigen Query-Sprachen

Aber XML hat auch Stärken! Die Query-Sprachen XPath und XQuery sind extrem mächtig. Mit XPath können Sie komplexe Pfade durch XML-Bäume navigieren - weit über das hinaus, was CSV je könnte. XQuery ist sogar Turing-vollständig! Das Problem: Die Lernkurve ist steil und die Tools sind kompliziert.

[https://learn.microsoft.com/en-us/previous-versions/windows/desktop/ms762271\(v=vs.85\)](https://learn.microsoft.com/en-us/previous-versions/windows/desktop/ms762271(v=vs.85))

🎯 **XPath Live-Demo:** Buchkatalog abfragen

Probieren Sie verschiedene XPath-Queries aus:

https://developer.mozilla.org/en-US/docs/Web/XML/XPath/Guides/Introduction_to_using_XPath_in_JavaScript

```

1 const xmlDoc = new DOMParser().parseFromString(books, 'text/xml');
2
3 // 🔎 Wählen Sie eine Query aus oder schreiben Sie eigene:
4
5 // Query 1: Alle Computer-Bücher
6 const query = '//book[genre="Computer"]/title/text()';
7
8 // Query 2: Bücher teurer als 10$
9 // const query = '//book[price > 10]/title/text()';
10
11 // Query 3: Alle Fantasy-Bücher von Eva Corets
12 // const query = '//book[author="Corets, Eva" and genre="Fantasy"]/ti
   /text()';
13
14 // Query 4: Günstigstes Buch (Titel)
15 // const query = '//book[price = min(/book/price)]/title/text()';
16
17 // Query 5: Durchschnittspreis aller Bücher
18 // const query = 'sum(/book/price) div count(/book)';
19
20 // Query 6: Bücher aus dem Jahr 2001
21 // const query = '//book[starts-with(publish_date, "2001")]/title/tex
22
23 // Query 7: Alle Genres (mit Duplikaten)
24 // const query = '//book/genre/text()';
25
26 // ✅ Query ausführen
27 const resultType = typeof query === 'string' &&
28     .... (query.includes('sum(') || query.includes('count(')
29     .... query.includes('avg(') || query.includes('div')))
30 ? XPathResult.NUMBER_TYPE
31 : XPathResult.ORDERED_NODE_SNAPSHOT_TYPE;
32
33 * const result = xmlDoc.evaluate(
34     query,
35     xmlDoc.documentElement,
36     null,
37     resultType,
38     null
39 );
40
41 // 📊 Ergebnis formatieren
42 let output;
43 * if (resultType === XPathResult.NUMBER_TYPE) {
44     output = `💰 Ergebnis: ${result.numberValue.toFixed(2)}`;
45 * } else {
46     const items = [];
47 *     for (let i = 0; i < result.snapshotLength; i++) {
48         items.push(`${i + 1}. ${result.snapshotItem(i).textContent}`);
49     }

```

```
50     output = items.length > 0
51     .... ? `📚 Gefunden: ${items.length} Ergebnisse\n\n${items.join('\n')}
52     .... : '❌ Keine Ergebnisse gefunden';
53 }
54
55 output;
```

Books.xml



```
1 <catalog>
2   <book id="bk101">
3     <author>Gambardella, Matthew</author>
4     <title>XML Developer's Guide</title>
5     <genre>Computer</genre>
6     <price>44.95</price>
7     <publish_date>2000-10-01</publish_date>
8     <description>An in-depth look at creating applications
9       with XML.</description>
10    </book>
11    <book id="bk102">
12      <author>Ralls, Kim</author>
13      <title>Midnight Rain</title>
14      <genre>Fantasy</genre>
15      <price>5.95</price>
16      <publish_date>2000-12-16</publish_date>
17      <description>A former architect battles corporate zombies,
18       an evil sorceress, and her own childhood to become queen
19       of the world.</description>
20    </book>
21    <book id="bk103">
22      <author>Corets, Eva</author>
23      <title>Maeve Ascendant</title>
24      <genre>Fantasy</genre>
25      <price>5.95</price>
26      <publish_date>2000-11-17</publish_date>
27      <description>After the collapse of a nanotechnology
28       society in England, the young survivors lay the
29       foundation for a new society.</description>
30    </book>
31    <book id="bk104">
32      <author>Corets, Eva</author>
33      <title>Oberon's Legacy</title>
34      <genre>Fantasy</genre>
35      <price>5.95</price>
36      <publish_date>2001-03-10</publish_date>
37      <description>In post-apocalypse England, the mysterious
38       agent known only as Oberon helps to create a new life
39       for the inhabitants of London. Sequel to Maeve
40       Ascendant.</description>
41    </book>
42    <book id="bk105">
43      <author>Corets, Eva</author>
44      <title>The Sundered Grail</title>
45      <genre>Fantasy</genre>
46      <price>5.95</price>
47      <publish_date>2001-09-10</publish_date>
48      <description>The two daughters of Maeve, half-sisters,
```

```
49      battle one another for control of England. Sequel to
50      Oberon's Legacy.</description>
51  </book>
52  <book id="bk106">
53      <author>Randall, Cynthia</author>
54      <title>Lover Birds</title>
55      <genre>Romance</genre>
56      <price>4.95</price>
57      <publish_date>2000-09-02</publish_date>
58      <description>When Carla meets Paul at an ornithology
59      conference, tempers fly as feathers get ruffled.</description>
60  </book>
61  <book id="bk107">
62      <author>Thurman, Paula</author>
63      <title>Splish Splash</title>
64      <genre>Romance</genre>
65      <price>4.95</price>
66      <publish_date>2000-11-02</publish_date>
67      <description>A deep sea diver finds true love twenty
68      thousand leagues beneath the sea.</description>
69  </book>
70  <book id="bk108">
71      <author>Knorr, Stefan</author>
72      <title>Creepy Crawlies</title>
73      <genre>Horror</genre>
74      <price>4.95</price>
75      <publish_date>2000-12-06</publish_date>
76      <description>An anthology of horror stories about roaches,
77      centipedes, scorpions and other insects.</description>
78  </book>
79  <book id="bk109">
80      <author>Kress, Peter</author>
81      <title>Paradox Lost</title>
82      <genre>Science Fiction</genre>
83      <price>6.95</price>
84      <publish_date>2000-11-02</publish_date>
85      <description>After an inadvertant trip through a Heisenberg
86      Uncertainty Device, James Salway discovers the problems
87      of being quantum.</description>
88  </book>
89  <book id="bk110">
90      <author>O'Brien, Tim</author>
91      <title>Microsoft .NET: The Programming Bible</title>
92      <genre>Computer</genre>
93      <price>36.95</price>
94      <publish_date>2000-12-09</publish_date>
95      <description>Microsoft's .NET initiative is explored in
96      detail in this deep programmer's reference.</description>
97  </book>
98  <book id="bk111">
99      <author>O'Brien, Tim</author>
```

```

...
100 <author>Galos, Mike</author>
101 <title>MSXML3: A Comprehensive Guide</title>
102 <genre>Computer</genre>
103 <price>36.95</price>
104 <publish_date>2000-12-01</publish_date>
105 <description>The Microsoft MSXML3 parser is covered in
106 detail, with attention to XML DOM interfaces, XSLT processing,
107 SAX and more.</description>
108 </book>
109 <book id="bk112">
110   <author>Galos, Mike</author>
111   <title>Visual Studio 7: A Comprehensive Guide</title>
112   <genre>Computer</genre>
113   <price>49.95</price>
114   <publish_date>2001-04-16</publish_date>
115   <description>Microsoft Visual Studio 7 is explored in depth,
116 looking at how Visual Basic, Visual C++, C#, and ASP+ are
117 integrated into a comprehensive development
118 environment.</description>
119 </book>

```

 Gefunden: 4 Ergebnisse

1. XML Developer's Guide
2. Microsoft .NET: The Programming Bible
3. MSXML3: A Comprehensive Guide
4. Visual Studio 7: A Comprehensive Guide

```

1 const xsltString = ` 
2 <xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL
3   /Transform">
4   <xsl:template match="/">
5     <h2>Katalog</h2>
6     <ul>
7       <xsl:for-each select="catalog/book[price &gt; 10]">
8         <li>
9           <b><xsl:value-of select="title"/></b>
10          - <xsl:value-of select="author"/> -
11          <xsl:value-of select="price"/> €
12        </li>
13      </xsl:for-each>
14    </ul>
15  </xsl:template>
16 </xsl:stylesheet>`;
17
18 // XML & XSLT parsen
19 const parser = new DOMParser();
20 const xml = parser.parseFromString(xmlString, "text/xml");

```

```
20 const xslt = parser.parseFromString(xsltString, "text/xml");
21
22 // Transformation
23 const processor = new XSLTProcessor();
24 processor.importStylesheet(xslt);
25 const fragment = processor.transformToFragment(xml, document);
26 const serializer = new XMLSerializer();
27 const htmlString = serializer.serializeToString(fragment);
28
29 console.html(htmlString);
```

Books.xml



```
1 <catalog>
2   <book id="bk101">
3     <author>Gambardella, Matthew</author>
4     <title>XML Developer's Guide</title>
5     <genre>Computer</genre>
6     <price>44.95</price>
7     <publish_date>2000-10-01</publish_date>
8     <description>An in-depth look at creating applications
9       with XML.</description>
10    </book>
11    <book id="bk102">
12      <author>Ralls, Kim</author>
13      <title>Midnight Rain</title>
14      <genre>Fantasy</genre>
15      <price>5.95</price>
16      <publish_date>2000-12-16</publish_date>
17      <description>A former architect battles corporate zombies,
18       an evil sorceress, and her own childhood to become queen
19       of the world.</description>
20    </book>
21    <book id="bk103">
22      <author>Corets, Eva</author>
23      <title>Maeve Ascendant</title>
24      <genre>Fantasy</genre>
25      <price>5.95</price>
26      <publish_date>2000-11-17</publish_date>
27      <description>After the collapse of a nanotechnology
28       society in England, the young survivors lay the
29       foundation for a new society.</description>
30    </book>
31    <book id="bk104">
32      <author>Corets, Eva</author>
33      <title>Oberon's Legacy</title>
34      <genre>Fantasy</genre>
35      <price>5.95</price>
36      <publish_date>2001-03-10</publish_date>
37      <description>In post-apocalypse England, the mysterious
38       agent known only as Oberon helps to create a new life
39       for the inhabitants of London. Sequel to Maeve
40       Ascendant.</description>
41    </book>
42    <book id="bk105">
43      <author>Corets, Eva</author>
44      <title>The Sundered Grail</title>
45      <genre>Fantasy</genre>
46      <price>5.95</price>
47      <publish_date>2001-09-10</publish_date>
48      <description>The two daughters of Maeve, half-sisters,
```

```
49      battle one another for control of England. Sequel to
50      Oberon's Legacy.</description>
51  </book>
52  <book id="bk106">
53      <author>Randall, Cynthia</author>
54      <title>Lover Birds</title>
55      <genre>Romance</genre>
56      <price>4.95</price>
57      <publish_date>2000-09-02</publish_date>
58      <description>When Carla meets Paul at an ornithology
59      conference, tempers fly as feathers get ruffled.</description>
60  </book>
61  <book id="bk107">
62      <author>Thurman, Paula</author>
63      <title>Splish Splash</title>
64      <genre>Romance</genre>
65      <price>4.95</price>
66      <publish_date>2000-11-02</publish_date>
67      <description>A deep sea diver finds true love twenty
68      thousand leagues beneath the sea.</description>
69  </book>
70  <book id="bk108">
71      <author>Knorr, Stefan</author>
72      <title>Creepy Crawlies</title>
73      <genre>Horror</genre>
74      <price>4.95</price>
75      <publish_date>2000-12-06</publish_date>
76      <description>An anthology of horror stories about roaches,
77      centipedes, scorpions and other insects.</description>
78  </book>
79  <book id="bk109">
80      <author>Kress, Peter</author>
81      <title>Paradox Lost</title>
82      <genre>Science Fiction</genre>
83      <price>6.95</price>
84      <publish_date>2000-11-02</publish_date>
85      <description>After an inadvertant trip through a Heisenberg
86      Uncertainty Device, James Salway discovers the problems
87      of being quantum.</description>
88  </book>
89  <book id="bk110">
90      <author>O'Brien, Tim</author>
91      <title>Microsoft .NET: The Programming Bible</title>
92      <genre>Computer</genre>
93      <price>36.95</price>
94      <publish_date>2000-12-09</publish_date>
95      <description>Microsoft's .NET initiative is explored in
96      detail in this deep programmer's reference.</description>
97  </book>
98  <book id="bk111">
99      <author>O'Brien, Tim</author>
```

```
  ...
100 <book>
101   <author>Galos, Mike</author>
102   <title>MSXML3: A Comprehensive Guide</title>
103   <genre>Computer</genre>
104   <price>36.95</price>
105   <publish_date>2000-12-01</publish_date>
106   <description>The Microsoft MSXML3 parser is covered in
107     detail, with attention to XML DOM interfaces, XSLT processing,
108     SAX and more.</description>
109 </book>
110 <book id="bk112">
111   <author>Galos, Mike</author>
112   <title>Visual Studio 7: A Comprehensive Guide</title>
113   <genre>Computer</genre>
114   <price>49.95</price>
115   <publish_date>2001-04-16</publish_date>
116   <description>Microsoft Visual Studio 7 is explored in depth,
117     looking at how Visual Basic, Visual C++, C#, and ASP+ are
118     integrated into a comprehensive development
119   environment.</description>
</book>
</catalog>
```

Katalog

XML Developer's Guide

– Gambardella, Matthew –
44.95 €

Microsoft .NET: The Programming Bible

– O'Brien, Tim –
36.95 €

MSXML3: A Comprehensive Guide

– O'Brien, Tim –
36.95 €

Visual Studio 7: A Comprehensive Guide

– Galos, Mike –
49.95 €

XPath - Navigation durch XML-Bäume:

```
/* Alle Buchtitel */  
//book/title  
  
/* Bücher nach 1980 */  
//book[year > 1980]/title  
  
/* Bücher von Kernighan */  
//book[authors/author = "Brian Kernighan"]/title  
  
/* Bücher über $50 */  
//book[price > 50]/@isbn  
  
/* Zweites Buch */  
//book[2]
```



```
//books[author]/*  
/* Bücher mit mehr als einem Autor */  
//book[count(authors/author) > 1]
```

XQuery - SQL für XML:

```
for $book in //book  
where $book/year > 1980 and $book/price < 100  
order by $book/price descending  
return  
  <result>  
    <title>{$book/title/text()}</title>  
    <cost>{$book/price/text()}</cost>  
  </result>
```



Vergleich:

- **CSV**: Keine Query-Sprache (nur externe Tools)
- **JSON**: JSONPath (limitiert, nicht standardisiert)
- **XML**: XPath/XQuery (mächtig, aber komplex)

Das Problem: Die Macht von XQuery rechtfertigt selten den XML-Overhead.

Warum existiert XML noch? Legacy! SOAP-APIs aus den 2000ern, Microsoft Office-Formate, SVG-Grafiken, RSS-Feeds - überall wo Kompatibilität wichtiger ist als Effizienz. Aber neue Projekte? Die starten mit JSON, nicht XML. Das ist die Lektion: Perfektion ist der Feind des Guten.

🎯 Wann XML verwenden - Wann nicht

XML ist perfekt für:

- **Legacy-Systeme** mit SOAP-APIs (keine Wahl)
- **Dokumenten-Workflows** mit komplexer Validation (DocBook, DITA)
- **Office-Formate** (.docx, .xlsx sind ZIP-Archive mit XML)
- **SVG/MathML** - Grafiken und Formeln als Vektordaten
- **RSS/Atom Feeds** - etablierter Standard
- **Konfigurationsdateien** mit Schema-Validierung (Maven pom.xml, ANT)

XML ist schlecht für:

- REST-APIs (JSON ist 3-6x effizienter)
- Real-time Kommunikation (Parsing-Overhead zu hoch)
- Mobile Apps (Bandbreite & Battery verschwendet)
- NoSQL-Datenbanken (JSON-nativer)
- Microservices (zu verbose für Service-Mesh)

Professor Freinets Regel: XML nur wo MÜSSEN (Legacy), nie wo KÖNNEN (neue Projekte)!

Zum Abschluss eine historische Perspektive: XML war die Antwort auf CSV-Chaos in den 90ern. Es brachte Ordnung, aber zum Preis der Praktikabilität. JSON lernte aus XMLs Fehlern: genug Struktur für Maschinen, genug Lesbarkeit für Menschen. Das ist Evolution in Reinform - und wir sind dabei, sie zu beobachten.

📋 Die XML-Lektion für Architekten

Was XML richtig machte:

- ✓ Hierarchische Struktur (besser als flaches CSV)
- ✓ Schema-Validierung (Integrität!)
- ✓ Namespaces (Kollisions-Vermeidung)
- ✓ Mächtige Query-Sprachen

Was XML falsch machte:

- ✗ Zu verbose (3-6x Overhead)
- ✗ Zu komplex (XSD ist monströs)
- ✗ Parsing-Performance (10-50x langsamer als JSON)
- ✗ Schlechte Developer-Experience

Die Architektur-Lektion:

„Perfektion ist der Feind des Guten.“

> XML versuchte, ALLES richtig zu machen - und wurde dadurch für VIELES unbrauchbar.

JSON gewann nicht durch Perfektion, sondern durch:

- Einfachheit (5 Minuten zum Lernen)
- Effizienz (nativ in JavaScript)
- Pragmatismus (gut genug für 95% der Fälle)

Das ist die Lektion für Datenbankarchitektur: Optimiere für den Normalfall, nicht den Extremfall.

JSON - Das Goldlöckchen-Format



JSON - JavaScript Object Notation - ist das Format, das den Sweet Spot zwischen CSV und XML gefunden hat. Nicht zu simpel, nicht zu komplex - genau richtig. Als praxisorientierter Architekt sage ich Ihnen: JSON ist das Format, das Sie in 5 Minuten lernen und für die nächsten 10 Jahre nutzen werden.

Was ist JSON eigentlich?

JSON steht für **JavaScript Object Notation** - ein textbasiertes Format für **strukturierte Daten** mit minimalem Overhead.

Erfunden: 2001 von Douglas Crockford

Ursprung: Subset von JavaScript (aber sprachunabhängig!)

Philosophie: „*So einfach wie möglich, aber nicht einfacher*“

Die 5-Minuten-Garantie: Nach diesem Abschnitt können Sie JSON lesen, schreiben und verstehen!

Lassen Sie uns mit der Anatomie beginnen. JSON hat exakt SECHS Datentypen - keine mehr, keine weniger. Zahlen, Strings, Booleans, null, Arrays und Objekte. Das wars. Diese bewusste Beschränkung macht JSON lernbar, aber nicht limitiert.

🔍 JSON-Anatomie: Die 6 Datentypen

```
{  
  "string": "Hallo Welt",           // Text in Anführungszeichen  
  "number": 42,                    // Integer oder Float (kein Unterschied)  
  "boolean": true,                // true oder false (kleingeschrieben!)  
  "null": null,                  // Explizites "nichts"  
  "array": [1, 2, 3],              // Geordnete Liste  
  "object": {                     // Key-Value-Paare  
    "nested": "Verschachtelung möglich!"  
  }  
}
```

Wichtige Regeln:

- Keys MÜSSEN in doppelten Anführungszeichen: `"name"` ✓ nicht `name` ✗
- Strings nur mit doppelten Anführungszeichen: `"text"` ✓ nicht `'text'` ✗
- Keine trailing commas: `[1, 2, 3]` ✓ nicht `[1, 2, 3,]` ✗
- Keine Kommentare: JSON ist pures Datenformat (Kommentare = Parsing-Error!)

Jetzt wird es praktisch! Hier ist derselbe Titanic-Passagier in CSV, JSON und XML. Sehen Sie den Unterschied? CSV ist flach, XML ist verbose, JSON ist strukturiert aber lesbar. Beachten Sie: JSON kann Verschachtelung (Name-Objekt), Arrays (Tickets) und verschiedene Typen - alles was CSV nicht kann, ohne XMLs Overhead!

📊 Vergleich: CSV vs. JSON vs. XML (Titanic-Passagier)

CSV (flach, keine Struktur): ` csv PassengerId,Survived,Pclass,Name,Sex,Age,Fare,Cabin,Embarked 2,1,1,
,,Cumings, Mrs. John Bradley“,female,38,71.2833,C85,C

JSON (strukturiert, lesbar):

```
{  
    "passengerId": 2,  
    "survived": true,  
    "class": 1,  
    "name": {  
        "family": "Cumings",  
        "given": "Florence Briggs",  
        "title": "Mrs.",  
        "husband": "John Bradley"  
    },  
    "demographics": {  
        "sex": "female",  
        "age": 38  
    },  
    "ticket": {  
        "number": "PC 17599",  
        "fare": 71.28,  
        "cabin": "C85",  
        "embarked": "Cherbourg"  
    }  
}
```

XML (verbose, bürokratisch):

```
<passenger id="2">  
    <survived>true</survived>  
    <class>1</class>  
    <name family="Cumings" given="Florence Briggs" title="Mrs.">  
        <husband>John Bradley</husband>  
    </name>  
    <demographics sex="female" age="38"/>  
    <ticket number="PC 17599" fare="71.28" cabin="C85" embarked="Cherbourg"/>  
</passenger>
```

Size-Vergleich: CSV: 87B | JSON: 285B (3.3x) | XML: 312B (3.6x)

Merke: JSON hat mehr Overhead als CSV, aber **deutlich weniger** als XML - und dafür Struktur!
Ein kritischer Punkt: JSON hat keine Datumstypen! Das ist kein Fehler, sondern Design. JSON bleibt bewusst einfach und überlässt Interpretation der Anwendung. Daher sehen Sie Dates oft als ISO-Strings. Auch die Zahl 42 vs „42“ - JSON unterscheidet nicht zwischen Integer und Float, alles ist „number“. Das vereinfacht Parser, aber Typsicherheit kommt von außen (TypeScript, JSON Schema).

JSON-Fallen: Was fehlt?

1. Keine Datumstypen:

```
{  
    "date": "2023-10-17",           // String, kein Date!  
    "timestamp": 1697500800,        // Unix-Timestamp als Zahl  
    "iso": "2023-10-17T10:00:00Z"   // ISO 8601 String (üblich)  
}
```

2. Keine Integer vs. Float Unterscheidung:

```
{  
    "integer": 42,      // Beides ist "number"  
    "float": 42.0,     // Parser entscheidet!  
    "scientific": 4.2e1 // Auch valid  
}
```

3. Keine Binärdaten:

```
{  
    "image": "base64encodedstring..." // Muss als String kodiert werden  
}
```

4. Keine Kommentare:

```
{  
    // "comment": "Das ist ein Error!" X  
    "_comment": "Workaround: Fake-Key" ✓  
}
```

5. Encoding muss UTF-8 sein: - JSON Standard erlaubt NUR UTF-8 (oder UTF-16/UTF-32) - Kein Latin-1, kein Windows-1252 → weniger Chaos als bei CSV!

Jetzt die gute Nachricht: JSON ist unglaublich praktisch in der Praxis. Jede moderne Programmiersprache hat native oder near-native Support. JavaScript? Es ist literale Syntax! Python? Ein dict ist schon JSON. Parsing ist 10-50x schneller als XML. Und Developer Experience? Sublime - keine XSD-Monster, keine DTDs, einfach schreiben und fertig.

✓ JSON in der Praxis: Warum es dominiert

Native JavaScript-Integration:

```
1 // JSON ist literale JavaScript-Syntax!  
2 const person = {  
3     "name": "Alice",  
4     "age": 30  
5 };  
6  
7 // Parsing & Serialisierung built-in  
8 const jsonString = JSON.stringify(person).
```

```

8 const jsonString = JSON.stringify(person),
9 const parsed = JSON.parse(jsonString);
10
11 console.log(jsonString);

```

```
{"name": "Alice", "age": 30}
```

Python (near-native):

```

1 import json
2
3 # Dict → JSON
4 data = {"name": "Alice", "age": 30}
5 json_str = json.dumps(data)
6
7 # JSON → Dict
8 parsed = json.loads(json_str)
9
10 print(json_str)

```

REST-APIs Standard:

- 95% aller modernen APIs nutzen JSON
- **Content-Type: application/json**
- Kleinere Payloads als XML (3x Faktor)

NoSQL-Datenbanken:

- MongoDB speichert BSON (Binary JSON)
- CouchDB, RethinkDB, Firebase - alle JSON-nativ
- Queries direkt auf JSON-Struktur

Die JSON-Erfolgsformel in einem Satz: Es ist einfach genug, dass Sie es in 5 Minuten lernen, aber mächtig genug für komplexe APIs. Kein Zufall, dass REST-APIs JSON nutzen, nicht XML. Developer Experience schlägt formale Perfektion - das ist die Lektion!

🎯 Wann JSON verwenden - Wann nicht

JSON ist perfekt für:

- REST-APIs (de facto Standard seit 2010)
- Web-Anwendungen (native JavaScript-Integration)
- NoSQL-Datenbanken (MongoDB, CouchDB, Firebase)
- Konfigurationsdateien (package.json, tsconfig.json)
- Data Science (wenn Struktur wichtiger als Size)
- Microservices-Kommunikation (schnell, kompakt)

JSON ist schlecht für:

- Binärdaten (Base64-Overhead 33%)
- Sehr große Datasets (CSV ist 3x kompakter)
- Streaming-Daten (keine partielle Parsing-Unterstützung)
- Komplexe Validierung (JSON Schema ist optional, nicht native)
- Kommentare notwendig (YAML oder TOML besser)

JSON ist OK, aber nicht optimal für:

- Human-Editing (YAML ist lesbarer ohne Quotes/Commas)
- Typsichere APIs (Protobuf oder gRPC besser)
- Extreme Performance (MessagePack, BSON schneller)

Professor Freinets Regel: JSON für alles wo Menschen UND Maschinen lesen - das sind 90% aller Fälle!

Zum Abschluss: JSON ist kein Zufall - es ist Evolution. CSV war zu simpel, XML zu komplex, JSON fand die Balance. Das ist die Architektur-Lektion: Erfolgreiche Technologien optimieren nicht für Perfektion, sondern für Adoption. JSON gewann, weil es einfach genug war, dass jeder Entwickler es ohne Handbuch versteht. Das ist der Benchmark für alle zukünftigen Datenformate!

Die JSON-Lektion für Architekten

Was JSON richtig machte:

-  **Einfachheit:** 6 Datentypen, 5 Minuten Lernzeit
-  **Lesbarkeit:** Selbsterklärend ohne Schema
-  **Performance:** 10-50x schneller als XML
-  **Adoption:** Native in JavaScript, near-native überall
-  **Pragmatismus:** Gut genug für 95% der Fälle

Was JSON bewusst opferte:

- ✗ Keine Kommentare (Daten ≠ Dokumentation)
- ✗ Keine Dates (Interpretation bleibt bei Anwendung)
- ✗ Kein natives Schema (JSON Schema optional)
- ✗ Keine Binärdaten (Base64-Workaround)

Die Architektur-Lektion:

„Simplicity is the ultimate sophistication.“ (Leonardo da Vinci)

> JSON gewann nicht durch Features, sondern durch **Fehlende Komplexität**.

Erfolgsformel:

1. Developer Experience > Formale Perfektion
2. Adoption > Theoretische Überlegenheit
3. „Gut genug“ > „Alles richtig“

Das ist die DNA erfolgreicher Technologien: Optimiere für den Normalfall, akzeptiere Schwächen im Extremfall.

```

1 fetch('https://restcountries.com/v3.1/name/germany')
2 //fetch('https://jsonplaceholder.typicode.com/users/1')
3 //fetch('https://api.github.com/repos/microsoft/vscode')
4 //fetch('https://api.open-meteo.com/v1/forecast?latitude=52.52&longitude=-41&current=temperature_2m')
5   .then(response => response.json())
6   .then(data => console.log(JSON.stringify(data, null, 2)))
i 7   .catch(error => console.error("Ups", error.message))

```

```
[  
  {  
    "name": {  
      "common": "Germany",  
      "official": "Federal Republic of Germany",  
      "nativeName": {  
        "deu": {  
          "official": "Bundesrepublik Deutschland",  
          "common": "Deutschland"  
        }  
      }  
    },  
    "tld": [  
      ".de"  
    ],  
    "cca2": "DE",  
    "ccn3": "276",  
    "cioc": "GER",  
    "independent": true,  
    "status": "officially-assigned",  
    "unMember": true,  
    "currencies": {  
      "EUR": {  
        "symbol": "€",  
        "name": "euro"  
      }  
    },  
    "idd": {  
      "root": "+4",  
      "suffixes": [  
        "9"  
      ]  
    },  
    "capital": [  
      "Berlin"  
    ],  
    "altSpellings": [  
      "DE",  
      "Federal Republic of Germany",  
      "Bundesrepublik Deutschland"  
    ],  
    "region": "Europe",  
    "subregion": "Western Europe",  
  }
```

```
"languages": {
    "deu": "German"
},
"latlng": [
    51,
    9
],
"landlocked": false,
"borders": [
    "AUT",
    "BEL",
    "CZE",
    "DNK",
    "FRA",
    "LUX",
    "NLD",
    "POL",
    "CHE"
],
"area": 357114,
"demonyms": {
    "eng": {
        "f": "German",
        "m": "German"
    },
    "fra": {
        "f": "Allemande",
        "m": "Allemand"
    }
},
"cca3": "DEU",
"translations": {
    "ara": {
        "official": "جمهورية ألمانيا الاتحادية",
        "common": "ألمانيا"
    },
    "bre": {
        "official": "Republik Kevreadel Alamagn",
        "common": "Alamagn"
    },
    "ces": {
        "official": "Spolková republika Německo",
        "common": "Německo"
    }
}
```

```
},
"cym": {
  "official": "Federal Republic of Germany",
  "common": "Germany"
},
"deu": {
  "official": "Bundesrepublik Deutschland",
  "common": "Deutschland"
},
"est": {
  "official": "Saksamaa Liitvabariik",
  "common": "Saksamaa"
},
"fin": {
  "official": "Saksan liittotasavalta",
  "common": "Saksa"
},
"fra": {
  "official": "République fédérale d'Allemagne",
  "common": "Allemagne"
},
"hrv": {
  "official": "Njemačka Federativna Republika",
  "common": "Njemačka"
},
"hun": {
  "official": "Német Szövetségi Köztársaság",
  "common": "Németország"
},
"ind": {
  "official": "Republik Federal Jerman",
  "common": "Jerman"
},
"ita": {
  "official": "Repubblica federale di Germania",
  "common": "Germania"
},
"jpn": {
  "official": "\u65e5\u672f\u8005\u5316",
  "common": "\u65e5\u672f"
},
"kor": {
  "official": "\uadcc \uacfc \uacfc\ud55c",
  "common": "\uacfc\ud55c"
},
```

```
    "common": "المانيا"
},
"nld": {
    "official": "Bondsrepubliek Duitsland",
    "common": "Duitsland"
},
"per": {
    "official": "جمهوري فدرال آلمان",
    "common": "آلمان"
},
"pol": {
    "official": "Republika Federalna Niemiec",
    "common": "Niemcy"
},
"por": {
    "official": "República Federal da Alemanha",
    "common": "Alemanha"
},
"rus": {
    "official": "Федеративная Республика Германия",
    "common": "Германия"
},
"slk": {
    "official": "Nemecká spolková republika",
    "common": "Nemecko"
},
"spa": {
    "official": "República Federal de Alemania",
    "common": "Alemania"
},
"srp": {
    "official": "Савезна Република Немачка",
    "common": "Немачка"
},
"swe": {
    "official": "Förbundsrepubliken Tyskland",
    "common": "Tyskland"
},
"tur": {
    "official": "Almanya Federal Cumhuriyeti",
    "common": "Almanya"
},
"urd": {
```

```
        "official": "وفاقی جمیوپریا جرمنی",
        "common": "جرمنی"
    },
    "zho": {
        "official": "\u5e02\u5316\u5316\u53f2\u5316",
        "common": "\u5e02\u5316"
    },
    "flag": "\u26f3\ufe0f",
    "maps": {
        "googleMaps": "https://goo.gl/maps/mD9FBMq1nvXUBrkv6",
        "openStreetMaps": "https://www.openstreetmap.org/relation/51477"
    },
    "population": 83491249,
    "gini": {
        "2016": 31.9
    },
    "fifa": "GER",
    "car": {
        "signs": [
            "DY"
        ],
        "side": "right"
    },
    "timezones": [
        "UTC+01:00"
    ],
    "continents": [
        "Europe"
    ],
    "flags": {
        "png": "https://flagcdn.com/w320/de.png",
        "svg": "https://flagcdn.com/de.svg",
        "alt": "The flag of Germany is composed of three equal horizontal bands of black, red and gold."
    },
    "coatOfArms": {
        "png": "https://mainfacts.com/media/images/coats_of_arms/de.png",
        "svg": "https://mainfacts.com/media/images/coats_of_arms/de.svg"
    },
    "startOfWeek": "monday",
    "capitalInfo": {
        "latlong": [

```

```
        52.52,  
        13.4  
    ]  
,  
  "postalCode": {  
    "format": "#####",  
    "regex": "^(\d{5})$"  
  }  
}  
]
```

YAML - JSON für Menschen

YAML - YAML Ain't Markup Language - ist JSONs menschenfreundlicher Cousin. Es wurde 2001 entwickelt, kurz nach JSON, mit einem klaren Ziel: Konfigurationsdateien, die Menschen gerne schreiben. Weniger Syntax-Noise, mehr Lesbarkeit. Docker Compose, Kubernetes, GitHub Actions - überall wo Konfiguration king ist, finden Sie YAML.

Was ist YAML?

YAML = YAML Ain't Markup Language (rekursives Akronym)

Erfunden: 2001 (parallel zu JSON)

Philosophie: „Human-readable data serialization“

Superset von JSON: Jedes JSON ist valides YAML!

Der Kern-Unterschied zu JSON:

- ✗ Keine geschweiften Klammern 
- ✗ Keine eckigen Klammern  (optional)
- ✗ Keine Anführungszeichen für Strings (meist)
- ✗ Keine Kommas
- ✗ Einrückung definiert Struktur (wie Python!)
- ✗ Kommentare erlaubt ()

Lassen Sie uns den direkten Vergleich sehen. Hier ist dieselbe Kubernetes-Config in JSON und YAML. Sehen Sie den Unterschied? JSON ist voller Syntax-Noise - Klammern, Quotes, Kommas. YAML ist clean - reine Daten, minimale Syntax. Das ist der Grund, warum DevOps YAML liebt: Sie schreiben Konfiguration, keine Parsing-Anweisungen!

JSON vs. YAML - Derselbe Inhalt

JSON (verbose, viele Sonderzeichen):

```
{  
  "apiVersion": "v1",  
  ...  
}
```

```

"Kind": "Service",
"metadata": {
  "name": "my-service",
  "labels": {
    "app": "myapp"
  }
},
"spec": {
  "ports": [
    {
      "port": 80,
      "targetPort": 8080
    }
  ],
  "selector": {
    "app": "myapp"
  }
}
}

```

YAML (clean, lesbar):

```

apiVersion: v1
kind: Service
metadata:
  name: my-service
  labels:
    app: myapp
spec:
  ports:
    - port: 80
      targetPort: 8080
  selector:
    app: myapp

```

Unterschied:

- JSON: 253 Zeichen, 18 Zeilen mit Syntax-Noise
- YAML: 158 Zeichen, 12 Zeilen ohne Clutter
- 37% kompakter, deutlich lesbarer!

 **Merke:** YAML = JSON ohne Syntax-Overhead für menschliche Augen!

YAML hat aber auch Tücken! Einrückung mit Spaces ist Pflicht - ein Tab bricht alles. Mehrdeutigkeiten gibt es auch: „yes“, „no“, „on“, „off“ werden zu Booleans geparsed, nicht Strings. Und die Spec ist riesig - YAML 1.2 hat Features, die kaum jemand nutzt. Daher: YAML für Konfig-Dateien, JSON für APIs. Das ist die Faustregel!

YAML-Fallen & Best Practices

1. Einrückung ist kritisch:

```
#  RICHTIG (2 Spaces)
parent:
    child: value

#  FALSCH (Tabs oder inkonsistente Spaces)
parent:
    child: value # 4 Spaces → Parsing-Error möglich!
```

2. Implizite Typen (können überraschen):

```
# Vorsicht: Diese werden zu Booleans!
boolean_yes: yes      # → true
boolean_no: no        # → false
boolean_on: on         # → true

# Strings explizit machen:
string_yes: "yes"     # → "yes" (String)
string_jaa: jaa        # → "no" (String)
```

3. Mehrzeilige Strings:

```
# | = Zeilenumbrüche behalten
description: |
  Dies ist eine
  mehrzeilige
  Beschreibung.

# > = Zeilenumbrüche werden zu Spaces
summary: >
  Langer Text der
  als eine Zeile
  behandelt wird.
```

4. Anker & Aliase (Wiederverwendung):

```
defaults: &default_settings
timeout: 30
retries: 3

service_a:
<<: *default_settings # Übernimmt defaults
name: ServiceA

service_b:
<<: *default_settings
name: ServiceB
```

Wann nutzen Sie YAML, wann JSON? Die Antwort ist einfach: YAML für Dateien, die Menschen schreiben und lesen - Konfiguration, CI/CD Pipelines, Infrastructure-as-Code. JSON für Maschinen - APIs, Datenbank-Export, programmatische Generierung. Das ist kein Zufall: Docker Compose, Kubernetes, Ansible, GitHub Actions - alles YAML. REST-APIs? Alle JSON. Die richtige Tool für den richtigen Job!

🎯 Wann YAML - Wann JSON?

✓ YAML ist perfekt für:

- **Konfigurationsdateien** (docker-compose.yml, .gitlab-ci.yml)
- **Infrastructure-as-Code** (Kubernetes Manifests, Ansible Playbooks)
- **CI/CD Pipelines** (GitHub Actions, CircleCI)
- **Menschliches Editing** (weniger Syntax-Fehler)
- **Dokumentation** (Kommentare erlaubt!)

✓ JSON ist besser für:

- **REST-APIs** (maschinenlesbar, schnelles Parsing)
- **Programmatische Generierung** (keine Einrückung-Probleme)
- **Datenbank-Export** (klar definierte Struktur)
- **Browser-Kommunikation** (native JavaScript-Unterstützung)

Die Faustregel:

Kriterium	YAML	JSON
Geschrieben von	Menschen	Maschinen
Gelesen von	Menschen	Maschinen
Use Case	Config	Data
Kommentare	✓ Ja	✗ Nein
Lesbarkeit	★★★★★	★★★
Parsing-Speed	★★	★★★★

Professor Freinets Regel: YAML für Config-Files die du editierst, JSON für Data-Transfer den Maschinen verarbeiten!

Zusammenfassung: YAML ist JSONs Antwort auf das Lesbarkeits-Problem. Es opfert Parsing-Speed für Developer-Experience. Das ist kein Bug, sondern Feature - denn Konfigurationsdateien werden einmal geschrieben, tausendmal gelesen. Optimiere für den Leser, nicht den Parser. Das ist die YAML-Philosophie in einem Satz!

📋 Die YAML-Lektion

YAML's Trade-offs:

- ✓ **Lesbarkeit:** Keine Syntax-Noise → 37% kompakter
- ✓ **Kommentare:** Dokumentation direkt in Config
- ✓ **DRY:** Anker/Aliase vermeiden Duplikation
- ✗ **Parsing:** 2-5x langsamer als JSON
- ✗ **Mehrdeutigkeiten:** `yes` / `no` werden zu Booleans
- ✗ **Einrückung:** Ein falscher Space bricht alles

Die Evolution:

CSV (1970er)	→ zu simpel
XML (1990er)	→ zu komplex
JSON (2001)	→ maschinenfreundlich
YAML (2001)	→ menschenfreundlich



Das Prinzip:

„Optimiere für die Häufigkeit der Operation.“

- > Config wird 1x geschrieben, 1000x gelesen → Optimiere für Leser!
- > API-Data wird 1000x generiert, 1000x geparsed → Optimiere für Parser!

YAML und JSON koexistieren, weil sie unterschiedliche Probleme lösen!

🎯 Fazit & Ausblick: Von Rohdaten zu strukturierten Systemen

Lassen Sie uns die Reise zusammenfassen. Wir haben heute die Grundlagen gelegt - nicht nur als historische Trivia, sondern als funktionale Analyse. Die DIKW-Pyramide zeigte uns: Daten ohne Kontext sind wertlos. Die historischen Beispiele zeigten: Jedes moderne DB-Feature hat einen Vorläufer. Und die Datenformate zeigten: Es gibt keinen Alleskönner - nur Trade-offs.

📚 Was Sie heute gelernt haben

1. Die DIKW-Pyramide als Analyserahmen:

- **Data:** Rohe Fakten (Federn zählen, Bytes speichern)
- **Information:** Kontextualisierte Daten (Kriegshäuptling-Status)
- **Knowledge:** Vernetzte Informationen (Gefahreneinschätzung)
- **Wisdom:** Handlungsfähigkeit (Entscheidung treffen)

→ Datenbanken bewegen sich zwischen Data und Information!

2. Historische DNA moderner Systeme:

- **Pacioli's Soll/Haben** → ACID-Transaktionen (Session L15!)
- **Karteikarten-Katalog** → B-Tree-Indizes (Session L16!)
- **Schiffstagebücher** → Write-Ahead-Logs (Session L22!)
- **Hollerith-Lochkarten** → Schema-Enforcement
- **Tontafeln** → Append-Only-Logs

→ Keine Konzepte sind wirklich neu - nur automatisiert!

4. Datenformate und ihre Trade-offs:

CSV:								→ Schnell, aber strukturlos
JSON:								→ Pragmatischer Allrounder
XML:								→ Perfekt, aber unflexibel
YAML:								→ JSON für Menschen

→ Kein Format ist „das Beste“ - nur „das Beste für X“!

Aber jetzt kommt der kritische Punkt: Alle diese Formate - CSV, JSON, XML, YAML - haben ein fundamentales Problem. Sie sind *Dateien*. Und Dateien sind dumm. Sie wissen nichts von Transaktionen, nichts von Indizes, nichts von Concurrency-Control. Wenn zwei Prozesse gleichzeitig schreiben? Datenverlust. Wenn Sie nach einem Feld suchen? Lineares Scannen. Wenn Sie Integrität garantieren wollen? Beten Sie!

Das fundamentale Problem: Dateien sind dumm

Was Dateien NICHT können:

- ✗ **Schnelles Suchen:** Linear Scan durch 1 GB CSV für eine Zeile?
- ✗ **Transaktionen:** Zwei gleichzeitige Writes → Einer verliert!
- ✗ **Indizes:** Jede Query liest ALLES ($O(n)$) statt $O(\log n)$)
- ✗ **Integrität:** Kein Foreign-Key-Check, keine Constraints
- ✗ **Concurrency:** Lock the whole file? Performance-Killer!
- ✗ **Recovery:** File korrupt? Alles weg!

Real-World Horror-Szenario:

```
# Zwei Prozesse schreiben gleichzeitig in users.csv
# Prozess A: Fügt "Alice" ein
# Prozess B: Fügt "Bob" ein
# Einheitswert: Daten werden überdeckt und verloren
```

Ergebnis: Datei korrupt, beide Einträge kaputt! ☀

Die Frage, die Sie sich stellen sollten:

„Wenn CSV so simpel ist und JSON so pragmatisch - warum brauchen wir überhaupt Datenbanken?“

Die Antwort: Weil Dateien **Datenhaltung** sind, aber keine **Datenverwaltung**!

Und hier beginnt die nächste Phase unserer Reise. Session L2 führt Key-Value Stores ein - das einfachste Datenbankparadigma. Denken Sie an ein gigantisches Hash-Map in Memory: Schlüssel → Wert, O(1) Zugriff, Transaktionen, Persistierung. Redis, Memcached, DynamoDB - Milliarden von Requests pro Tag. Aber auch sie haben Grenzen: Keine Queries, keine Relationen, keine Joins. Das ist die Evolution in Aktion!

🔮 Ausblick: Session L2 - Key-Value Stores

Die nächste Evolutionsstufe:

Wir haben gelernt: CSV ist zu simpel, XML ist zu komplex, JSON ist genau richtig für *Daten*.

Aber: Alle sind **Dateien ohne Verwaltung**!

Key-Value Stores lösen DAS Problem:

Problem	→ Lösung (Key-Value)	🔗
Langsames Suchen	→ Hash-Map (O(1))	
Keine Transaktionen	→ ACID-Garantien	
Kein Concurrency	→ Lock-free Reads	
Keine Persistierung	→ Write-Ahead-Log	
Linear Scan	→ Direkt-Zugriff	

Zum Abschluss die wichtigste Lektion: Datenbankarchitektur ist KEINE Religion. Es gibt kein „bestes“ System, nur „best für diesen Use-Case“. CSV für schnelle Exports. JSON für APIs. XML für Legacy. Und bald: Key-Value für Caching, Document für Flexibilität, Column für Analytics, Relational für Integrität, Graph für Beziehungen. Jedes Paradigma ist eine Antwort auf spezifische Schwächen des vorherigen. Das ist Evolution - und Sie sind jetzt in der Lage, sie zu analysieren!

🎓 Die Meta-Lektion: Trade-offs akzeptieren

Warum diese Vorlesung NICHT mit SQL startete:

Viele Datenbank-Kurse beginnen mit relationalen Systemen und SQL.

Das Problem: Sie verstehen nicht, *warum* Relational gut ist - nur *dass* es gut ist.

Unser Ansatz:

Rohdaten (CSV)	→ Verstehe das Problem	🔗
Key-Value (Redis)	→ Erste Lösung (schnell, simpel)	
Document (MongoDB)	→ Zweite Lösung (flexibel)	
(Wide) Column (Cassandra)	→ Dritte Lösung (analytisch)	
Relational (PostgreSQL)	→ Vierte Lösung (formal korrekt)	

Jedes System ist die Antwort auf die Schwächen des vorherigen!

Die 5 Vergleichsachsen als Kompass:

- Sie haben jetzt ein **Werkzeug** zum Analysieren
- Am Ende: **Vollständige Übersicht** aller Paradigmen
- Ziel: **Entscheidungsfähigkeit**, keine Dogmen!