Think BIGGER

DEEP DIVE FÜR FACHLEUTE

Datenbanken

Architektur, Speichermechanik, Performance und Datenbanktypen

Autor	Erhard Rainer
Veröffentlicht	04.04.2026
Aktualisiert	10.04.2026
Umfang	ca. 32 Min. Lesezeit
Online	www.thinkbigger.at/wissen/datenbanken/

thinkbigger.at

Business Intelligence · Machine Learning · KI-Agenten · Schulungen

Erhard Rainer · Schönaugasse 106 · 8020 Graz · info@think-bigger.at

Inhalt

0. Überblick 2
1. Was ist eine Datenbank? 2
2. Aus welchen Bausteinen besteht ein Datenbanksystem? 2
3. Welche Arten von Datenbanken gibt es? 4
4. Nicht jede Datenbankart beantwortet dieselbe Frage 10
5. SQL und ANSI-Standards 10
6. Wie werden Daten in Datenbanken gespeichert? 12
7. Warum ist der Zugriff auf Datenbanken so schnell, obwohl die Daten im Filesystem liegen? 16
8. B-Tree, Hash, Column Store und LSM: Warum die Speicherstruktur so viel ausmacht 19
9. Transaktionen, Konsistenz und Recovery 20
10. Datenqualität, ETL und Datenbanken 21
11. Cloud-Native und die Trennung von Compute und Storage 22
12. Lernen, Testen und Prototyping 23
13. Wichtige Datenbanksysteme im Überblick 23
14. Wie wählt man die richtige Datenbank aus? 24
15. Häufige Missverständnisse über Datenbanken 24
16. Fazit 25

0. Überblick

Eine Datenbank ist nicht einfach nur eine Datei mit Daten. Sie ist ein System, das Daten strukturiert speichert, Beziehungen und Regeln verwaltet, parallele Zugriffe koordiniert, Änderungen sicher protokolliert und Anfragen in planbarer Zeit beantwortet.

Wer verstehen will, warum Datenbanken so leistungsfähig sind, muss drei Ebenen auseinanderhalten:

1. Die fachliche Ebene: Welche Informationen gibt es, wie hängen sie zusammen, welche Regeln gelten?
2. Die logische Ebene: In welchen Tabellen, Dokumenten, Knoten, Spaltenfamilien oder Schlüssel-Wert-Paaren werden die Daten modelliert?
3. Die physische Ebene: In welchen Seiten, Segmenten, Indexstrukturen, Logdateien, Caches und Speicherblöcken landen diese Daten tatsächlich?

Genau diese drei Ebenen machen den Unterschied zwischen "Daten liegen irgendwo auf Platte" und "Daten sind kontrolliert, schnell und sicher abrufbar".

1. Was ist eine Datenbank?

Im engeren Sinn bezeichnet eine Datenbank den persistenten Datenbestand. Im Alltag meint man mit "Datenbank" aber fast immer das komplette Datenbankmanagementsystem, also die Software, die den Datenbestand verwaltet.

Ein Datenbankmanagementsystem übernimmt unter anderem:

- das Speichern und Laden der Daten,
- die Verwaltung von Schemata, Tabellen, Dokumenten oder Graphen,
- das Durchsetzen von Integritätsregeln,
- die Optimierung von Abfragen,
- die Absicherung paralleler Zugriffe,
- die Transaktionsverwaltung,
- die Wiederherstellung nach Abstürzen,
- die Rechte- und Benutzerverwaltung,
- das Monitoring und Tuning.

Eine gute Merkhilfe lautet:

> Eine Datei speichert Bytes. Eine Datenbank speichert Bedeutung.

Das ist der Kern: Eine Datenbank kennt Datentypen, Beziehungen, Indizes, Statistiken, Sichtbarkeitsregeln und Wiederanlaufmechanismen. Ein Dateisystem kennt all das nicht. Es speichert nur Blöcke in Dateien und Ordnern.

2. Aus welchen Bausteinen besteht ein Datenbanksystem?

Hinter einer scheinbar einfachen Anfrage wie `SELECT * FROM kunden WHERE kundennummer = 4711` arbeitet ein ganzer Stapel spezialisierter Komponenten.

2.1 Parser und Query Processor

Der Parser zerlegt SQL, CQL oder eine andere Abfragesprache in eine interne Repräsentation. Danach prüft das System:

- ist die Syntax gültig,

- existieren Tabelle, Collection oder View,
- passen Datentypen und Funktionen,
- darf der aufrufende Benutzer die Operation überhaupt ausführen.

2.2 Optimizer

Der Optimizer entscheidet, wie die Anfrage ausgeführt wird. Er wählt zum Beispiel:

- Vollscan oder Indexzugriff,
- Join-Reihenfolge,
- Hash Join, Merge Join oder Nested Loop,
- parallele oder serielle Ausführung,
- Sortierung im Speicher oder auf Platte.

Ohne Optimizer wäre selbst eine gute Datenstruktur oft langsam, weil das System den falschen Ausführungsweg wählen könnte.

2.3 Storage Engine

Die Storage Engine ist der Teil, der Daten tatsächlich auf Seitenebene organisiert. Sie entscheidet unter anderem:

- welche Seitengröße verwendet wird,
- wie Zeilen oder Spaltenblöcke aufgebaut sind,
- wie Indizes gespeichert werden,
- wie freie Bereiche gefunden werden,
- wie Änderungen protokolliert werden,
- wie Hintergrundprozesse wie Checkpoint, Vacuum, Compaction oder Delta Merge laufen.

2.4 Buffer Manager

Der Buffer Manager verwaltet den schnellen Arbeitsspeicher der Datenbank. Er hält häufig benötigte Seiten im RAM, damit bei einem Zugriff nicht jedes Mal erneut auf das Dateisystem zugegriffen werden muss.

2.5 Transaction Manager

Der Transaction Manager sorgt für ACID-Eigenschaften:

- Atomicity,
- Consistency,
- Isolation,
- Durability.

Er stellt also sicher, dass Änderungen vollständig oder gar nicht wirksam werden, parallele Prozesse sich nicht unkontrolliert in die Quere kommen und bestätigte Transaktionen nach einem Absturz wiederhergestellt werden können.

2.6 Logging und Recovery

Praktisch jede ernstzunehmende Datenbank führt ein Änderungsprotokoll. Dieses Protokoll heißt je nach System etwa:

- Write-Ahead Log,
- Redo Log,
- Undo Log,
- Transaction Log,
- Commit Log.

Das Protokoll ist zentral für Absturzsicherheit, Replikation und Wiederherstellung.

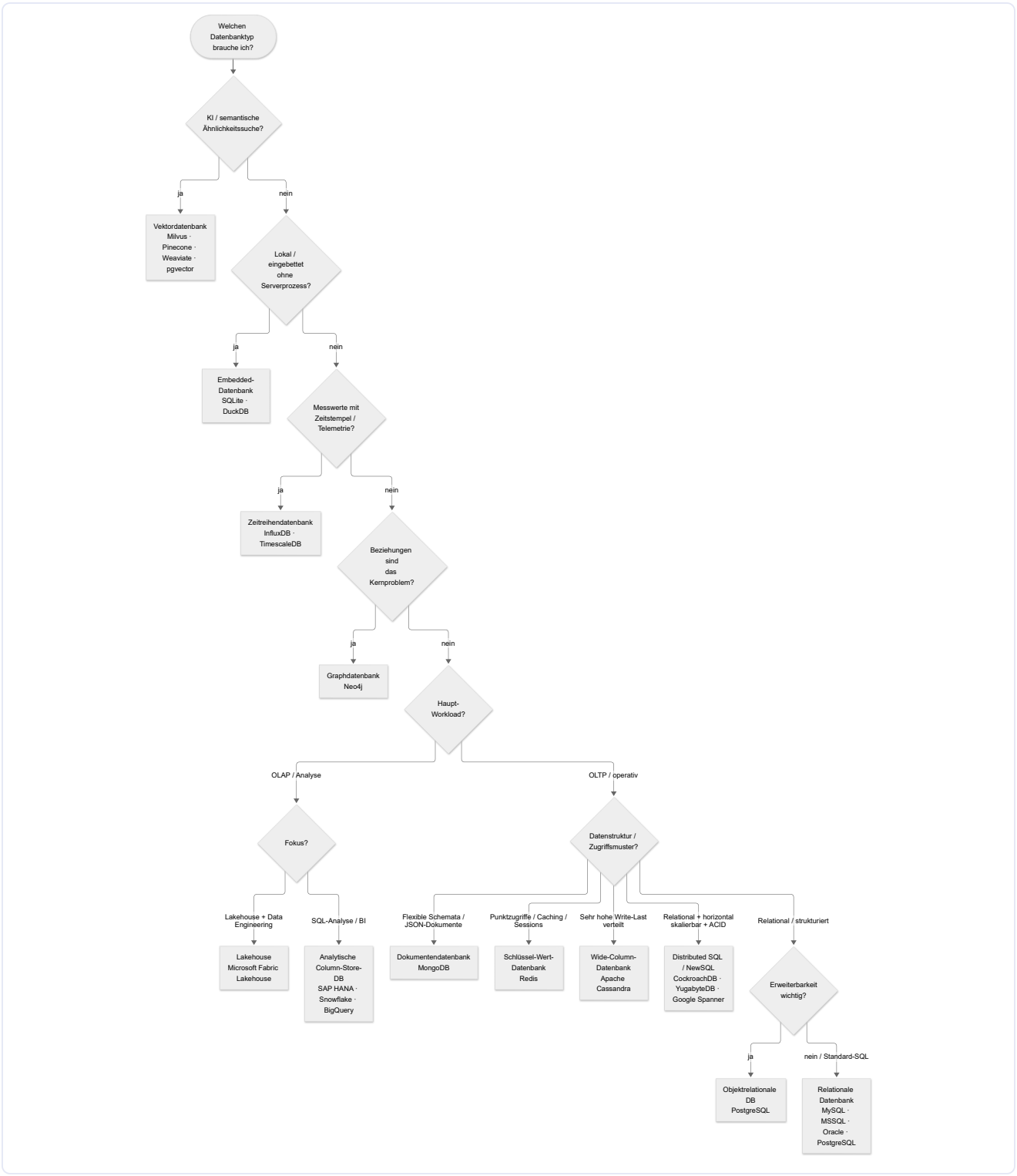
3. Welche Arten von Datenbanken gibt es?

Die Frage nach der "Art" einer Datenbank kann man auf drei unterschiedliche Arten beantworten:

1. Nach dem Datenmodell.
2. Nach dem Speicher- und Verarbeitungsmodell.
3. Nach dem Einsatzmuster.

Viele Missverständnisse entstehen, weil diese Ebenen vermischt werden. Eine Datenbank kann gleichzeitig relational, in-memory, spaltenorientiert und verteilt sein. Das schließt sich nicht aus.

Das folgende Entscheidungsdiagramm gibt einen Überblick über alle gängigen Datenbanktypen und hilft, ausgehend vom konkreten Bedarf den passenden Typ – und typische Vertreter – zu identifizieren:



3.1 Relationale Datenbanken

Relationale Datenbanken speichern Daten in Tabellen mit Zeilen und Spalten. Beziehungen werden über Schlüssel modelliert. Das relationale Modell ist stark, wenn Datenkonsistenz, klare Regeln, Joins, Transaktionen und strukturierte Abfragen wichtig sind.

Typische Merkmale:

- Tabellen, Primärschlüssel, Fremdschlüssel,
- SQL als Standardsprache,
- starke Integritätsregeln,

- sehr gute Eignung für OLTP,
- häufig auch gute analytische Funktionen.

Typische Vertreter:

- PostgreSQL
- MySQL
- MSSQL / Microsoft SQL Server
- Oracle Database

Wenn du das fachliche Entwerfen von Tabellen, Beziehungen und Normalisierung vertiefen willst, ist die Seite zur Datenbankmodellierung die logische Ergänzung.

3.2 Objektrelationale Datenbanken

Objektrelationale Datenbanken bauen auf dem relationalen Modell auf, erweitern es aber um zusätzliche Typen, Funktionen oder Erweiterbarkeit. PostgreSQL ist das klassische Beispiel: relational im Kern, aber stark erweiterbar bei Datentypen, Indizes, Geodaten, JSON und Funktionen.

3.3 Schlüssel-Wert-Datenbanken

Hier wird unter einem Schlüssel ein Wert gespeichert. Das Modell ist minimalistisch, aber extrem leistungsfähig für bestimmte Zugriffsmuster.

Typische Stärken:

- sehr schnelle Punktzugriffe,
- einfache horizontale Skalierung,
- gute Eignung für Caching, Session-Daten oder Zustandsverwaltung.

Typischer Vertreter:

- Redis

3.4 Dokumentenorientierte Datenbanken

Dokumentendatenbanken speichern zusammengehörige Informationen als Dokument, häufig in JSON- oder BSON-ähnlicher Form. Das ist stark, wenn Datenstrukturen flexibel sind oder sich häufig ändern.

Typische Merkmale:

- semistrukturierte Daten,
- verschachtelte Objekte und Arrays,
- flexible Schemata,
- gute Entwicklerergonomie.

Typischer Vertreter:

- MongoDB

3.5 Wide-Column- oder Column-Family-Datenbanken

Diese Systeme sind nicht mit analytischen Column Stores gleichzusetzen. Wide-Column-Datenbanken gruppieren Daten um Partitionen und Zugriffspfade. Sie sind oft auf große Cluster, hohe Verfügbarkeit und enorme Schreiblast ausgelegt.

Typischer Vertreter:

- Apache Cassandra

3.6 Graphdatenbanken

Graphdatenbanken modellieren Knoten, Kanten und Eigenschaften. Sie sind sinnvoll, wenn Beziehungen selbst der eigentliche Kern des Problems sind, etwa bei Betrugserkennung, Netzwerken, Routing oder Wissensgraphen.

Typischer Vertreter:

- Neo4j

3.7 Zeitreihendatenbanken

Zeitreihendatenbanken sind auf Messwerte mit Zeitstempel spezialisiert, etwa Telemetrie, Sensorik, Marktpreise oder Infrastrukturmetriken. Häufig optimieren sie:

- hohe Schreibfrequenzen,
- Kompression,
- Downsampling,
- Window Functions,
- TTL und automatische Datenalterung.

3.8 Analytische spaltenorientierte Datenbanken

Hier werden Daten spaltenweise statt zeilenweise gespeichert. Das ist ideal, wenn nur wenige Spalten aus sehr vielen Zeilen gelesen und aggregiert werden sollen.

Typische Stärken:

- sehr gute Kompression,
- wenig I/O für Analysen,
- schnelle Aggregationen,
- Vektorverarbeitung.

Ein klassisches Beispiel im SAP-Umfeld ist SAP HANA / HANADB.

3.9 In-Memory-Datenbanken

In-Memory-Datenbanken halten einen großen Teil des aktiven Datenbestands im RAM. Das bedeutet nicht, dass sie keine Persistenz haben. Es bedeutet, dass der primäre Arbeitsbereich der Hauptspeicher ist und die Persistenzmechanismen darum herum organisiert werden.

3.10 Embedded-Datenbanken

Embedded-Datenbanken laufen direkt in einer Anwendung oder lokal auf einem Gerät, oft ohne separaten Serverprozess. Sie sind stark bei lokalen Anwendungen, mobilen Geräten, Testumgebungen oder Edge-Szenarien.

Typische Vertreter:

- SQLite (klassisch, weit verbreitet in mobilen und Desktop-Applikationen)
- DuckDB (analytisch, eingebettet, stark bei SQL-Analysen direkt auf Datendateien)

3.11 Vektordatenbanken (KI & Machine Learning)

Mit dem Aufstieg von Large Language Models (LLMs) und generativer KI sind Vektordatenbanken ins Zentrum gerückt. Sie speichern Daten nicht primär als Text oder Zahl, sondern als hochdimensionale Vektoren (Embeddings), die semantische Ähnlichkeiten abbilden. Das ist der Schlüssel für moderne KI-Suchmechanismen und Retrieval-Augmented Generation (RAG).

Typische Stärken:

- schnelle Nearest-Neighbor-Suche (Ähnlichkeitssuche),
- Integration mit Transformer-Modellen und Embeddings,

- Kombination von Metadaten-Filtern und semantischer Suche.

Typische Vertreter:

- Milvus, Pinecone, Weaviate (spezialisierte Vektor-DBs)
- PostgreSQL mit der `pgvector`-Erweiterung

3.12 Distributed SQL (NewSQL)

Distributed SQL versucht, das Beste aus zwei Welten zu vereinen: die horizontalen Skalierungsfähigkeiten und Ausfallsicherheit von NoSQL-Systemen gepaart mit den starken ACID-Garantien und der standardisierten relationalen Struktur traditioneller SQL-Datenbanken. Die Daten werden über die Knoten hinweg verteilt und konsistent gehalten, ohne dass der Entwickler die Sharding-Logik in der Applikation nachbauen muss.

Typische Vertreter:

- CockroachDB
- YugabyteDB
- Google Cloud Spanner

3.13 SQL-Befehlsgruppen im Datenbanktyp-Vergleich

Die vier SQL-Befehlsgruppen DDL, DML, DCL und TCL sind nicht bei allen Datenbanktypen gleichermaßen verfügbar. Die folgende Matrix zeigt, welche Gruppen pro Datenbanktyp vollständig, eingeschränkt oder gar nicht vorhanden sind.

Legende:

- ✅ vollständig vorhanden (standardkonform oder mit geringen systematischen Einschränkungen)
- ⚠️ eingeschränkt oder nur in proprietärer Variante verfügbar
- ❌ nicht vorhanden

Datenbanktyp	DDL	DML	DCL	TCL
Relationale Datenbanken	✅ vollständig	✅ vollständig	✅ vollständig	✅ vollständig
Objektrelationale Datenbanken	✅ vollständig + CREATE TYPE, DOMAIN, EXTENSION	✅ vollständig	✅ vollständig	✅ vollständig
Schlüssel-Wert-Datenbanken	❌ kein Schema, kein SQL-DDL	⚠️ proprietär (SET, GET, HSET …)	⚠️ ACL-Modell vorhanden (kein GRANT/REVOKE)	⚠️ MULTI/EXEC statt COMMIT/ROLLBACK; kein SAVEPOINT
Dokumentenorientierte Datenbanken	⚠️ Collections werden implizit angelegt; Schema-Validation per JSON-Schema-API; kein SQL-DDL	⚠️ proprietäre API (find, insertOne, updateOne …); kein SQL; SQL-Translator in MongoDB Atlas ab v6	⚠️ Rollenkonzept vorhanden; kein SQL GRANT/REVOKE	⚠️ Multi-Document ACID-Transaktionen ab v4.0; COMMIT/ABORT vorhanden; kein SAVEPOINT
Wide-Column-Datenbanken	✅ CQL: CREATE/ALTER/DROP TABLE, KEYSPACE, INDEX vorhanden	⚠️ CQL ähnelt SQL (SELECT/INSERT/UPDATE/DELETE), aber keine JOINs, keine Subqueries, eingeschränkte WHERE-Bedingungen (nur Partition Key und Clustering Columns)	⚠️ GRANT/REVOKE in CQL vorhanden, aber nur bei explizit aktivierter Authentifizierung und Autorisierung	⚠️ Lightweight Transactions (Paxos-Protokoll) für bedingte Einzelzeilen-Operationen; kein BEGIN/COMMIT/ROLLBACK über mehrere Statements

Datenbanktyp	DDL	DML	DCL	TCL
Graphdatenbanken	⚠️ schema-optional; CREATE CONSTRAINT und CREATE INDEX per Cypher möglich; kein klassisches CREATE TABLE	⚠️ Cypher-Sprache (MATCH, CREATE, MERGE, SET, DELETE, REMOVE); kein SQL	⚠️ GRANT/DENY/REVOKE in Cypher ab Neo4j 4.0 vorhanden; proprietäre Syntax	⚠️ ACID-Transaktionen vorhanden; Steuerung über Client-API (beginTx, commit, rollback); kein SQL COMMIT/ROLLBACK in Cypher selbst
Zeitreihendatenbanken	⚠️ systemabhängig [1]	⚠️ systemabhängig [2]	⚠️ systemabhängig	❌ / ⚠️ systemabhängig [3]
Analytische Column-Store-DBs	✅ vollständig	✅ vorhanden; UPDATE/DELETE systemabhängig eingeschränkt [4]	✅ GRANT/REVOKE vorhanden	⚠️ systemabhängig [5]
In-Memory-Datenbanken	✅ / ❌ modellabhängig [6]	✅ / ⚠️ modellabhängig	✅ / ⚠️ modellabhängig	✅ / ⚠️ modellabhängig
Embedded-Datenbanken	✅ vorhanden [7]	✅ vollständig	❌ kein Benutzerkonzept (kein Mehrbenutzerbetrieb)	✅ vollständig (COMMIT, ROLLBACK, SAVEPOINT)
Vektordatenbanken	⚠️ Collection- und Index-Definition per SDK/API; kein SQL-DDL; pgvector: wie PostgreSQL	⚠️ proprietäre Such-API (Insert, ANN-Search); kein SQL; pgvector: SQL + Vektoroperatoren	⚠️ API-Key-Management und Zugriffsrollen; kein SQL GRANT/REVOKE; pgvector: wie PostgreSQL	❌ spezialisierte Vektor-DBs ohne Transaktionen; pgvector: vollständig wie PostgreSQL
Distributed SQL (NewSQL)	✅ vollständig; verteiltes DDL wird über alle Knoten koordiniert	✅ vollständig; ACID-Garantien gelten knotenübergreifend	✅ GRANT/REVOKE vollständig vorhanden	✅ vollständig (COMMIT, ROLLBACK, SAVEPOINT über verteilte Transaktionen)

Hinweise:

[1] **Zeitreihendatenbanken – DDL:** Stark systemabhängig. InfluxDB 2.x kennt kein SQL-DDL: Buckets, Measurements und Retention Policies werden per API oder CLI verwaltet. TimescaleDB ist eine PostgreSQL-Extension und bietet vollständiges SQL-DDL inklusive aller PostgreSQL-Erweiterungen.

[2] **Zeitreihendatenbanken – DML:** InfluxDB nutzt Flux (funktional, ab v2) oder InfluxQL (proprietär, SQL-ähnlich, ab v1/v1-Kompatibilität), Writes erfolgen über das Line Protocol. TimescaleDB: vollständiges PostgreSQL-SQL inklusive Window Functions und zeitreihentypischer Erweiterungen wie time_bucket().

[3] **Zeitreihendatenbanken – TCL:** InfluxDB besitzt kein klassisches Transaktionsmodell; Writes sind append-only und atomar auf Punkt-Ebene. TimescaleDB erbt vollständige ACID-Transaktionen von PostgreSQL.

[4] **Analytische Column-Store-DBs – DML:** BigQuery unterstützt UPDATE, DELETE und MERGE in Standard SQL, ist jedoch nicht für häufige Punkt-Updates oder kleinteilige DML-Workloads ausgelegt (hohe Latenz, Slot-Verbrauch). Snowflake bietet vollständiges DML inklusive MERGE. SAP HANA unterstützt vollständiges DML und ist hybrid für OLTP und OLAP ausgelegt.

[5] **Analytische Column-Store-DBs – TCL:** Snowflake: COMMIT, ROLLBACK und SAVEPOINT vollständig. BigQuery: Transaktionen für DML-Statements vorhanden (BEGIN TRANSACTION / COMMIT TRANSACTION), aber keine klassischen interaktiven Session-Transaktionen mit beliebigen Cursor-Operationen. SAP HANA: COMMIT/ROLLBACK vollständig, SAVEPOINT systemspezifisch.

[6] **In-Memory-Datenbanken:** Das Speichermedium (RAM als primärer Arbeitsbereich) ist unabhängig vom Datenmodell. Relationale In-Memory-Systeme wie VoltDB, MemSQL/SingleStore oder SAP HANA bieten vollständige SQL-Unterstützung in allen vier Bereichen. Cache-orientierte In-Memory-Systeme wie Redis folgen dem Schlüssel-Wert-Modell mit den dort beschriebenen Einschränkungen.

[7] **Embedded-Datenbanken – DDL:** SQLite unterstützt ALTER TABLE eingeschränkt: DROP COLUMN ist erst ab SQLite 3.35 (März 2021) verfügbar, RENAME COLUMN ab SQLite 3.25. ADD COLUMN ist seit Langem vorhanden. DuckDB bietet vollständiges DDL ohne diese Einschränkungen.

4. Nicht jede Datenbankart beantwortet dieselbe Frage

Zur Einordnung hilft eine kompakte Matrix:

Frage	Typische Antwort
"Ich brauche starke Transaktionen und konsistente Geschäftsprozesse."	Relationale Datenbank
"Ich brauche flexible Dokumente und schnelle Produktentwicklung."	Dokumentendatenbank
"Ich brauche hohe Verfügbarkeit über viele Knoten und extrem hohe Write-Last."	Wide-Column / verteilte NoSQL-DB
"Ich analysiere Milliarden Zeilen mit wenigen Spalten pro Query."	Column Store / analytische DB
"Meine eigentliche Logik steckt in Beziehungen zwischen Entitäten."	Graphdatenbank
"Ich speichere Messwerte oder Events über Zeit."	Zeitreihendatenbank
"Ich will Rohdaten, Dateien und Tabellen in einer Analytics-Plattform kombinieren."	Lakehouse

Die entscheidende Lehre lautet: Es gibt keine "beste" Datenbank im luftleeren Raum. Es gibt nur passende oder unpassende Systeme für einen bestimmten Workload, ein bestimmtes Datenmodell und bestimmte Betriebsanforderungen.

4.1 OLTP und OLAP: zwei sehr unterschiedliche Welten

Ein Grund, warum Datenbankdiskussionen oft aneinander vorbeigehen, ist die Vermischung von OLTP und OLAP.

Merkmal	OLTP	OLAP
Ziel	operative Geschäftsprozesse	Analyse, Reporting, Planung
typische Queries	kurz, selektiv, viele Einzeltransaktionen	lange, aggregierend, scan-lastig
Schreibverhalten	viele kleine Inserts/Updates	seltener, oft batch-orientiert
Datenmodell	stark normalisiert	häufig denormalisiert, Sternschema, breite Faktentabellen
wichtige Priorität	geringe Latenz, Konsistenz	Durchsatz, Scan-Leistung, Kompression
typische Systeme	PostgreSQL, SQL Server, MySQL	Snowflake, BigQuery, SAP HANA, klassische Warehouses

Relationale Kerndatenbanken sind oft stark im OLTP. Analytische Plattformen, Warehouses und viele Lakehouse-Architekturen sind dagegen klar OLAP-orientiert. Wer diese Unterscheidung nicht sauber trifft, wählt oft das falsche System für den falschen Workload.

5. SQL und ANSI-Standards

SQL (Structured Query Language) ist die Abfragesprache für relationale Datenbanken. Sie ist seit Jahrzehnten durch ANSI und ISO standardisiert und definiert ein gemeinsames Fundament, von dem alle gängigen Systeme in unterschiedlichem Ausmaß abweichen.

5.1 Entstehung von SQL

SQL entstand Anfang der 1970er-Jahre bei IBM, zunächst unter dem Namen SEQUEL (Structured English Query Language). Grundlage war das relationale Modell, das Edgar F. Codd 1970 beschrieben hatte. ANSI standardisierte SQL erstmals 1986, die ISO folgte unmittelbar danach.

5.2 Versionsgeschichte des Standards

Version	Jahr	Wichtigste Neuerungen
SQL-86 / SQL-87	1986/1987	Erster ANSI/ISO-Standard
SQL-89	1989	Integrity Enhancement Feature, kleinere Korrekturen
SQL-92 (SQL2)	1992	JOINs, Unterabfragen, CASE, neue Datentypen, Zeichensätze, drei Konformitätsstufen
SQL:1999 (SQL3)	1999	Trigger, rekursive Abfragen (`WITH RECURSIVE`), erweiterte Typen, Rollen
SQL:2003	2003	Window Functions, `MERGE`, XML-Erweiterung (SQL/XML), generierte Spalten
SQL:2006	2006	Vertiefung SQL/XML
SQL:2008	2008	`TRUNCATE`, `FETCH FIRST n ROWS ONLY`
SQL:2011	2011	Temporale Daten: Systemzeit, Anwendungszeit, Perioden
SQL:2016	2016	JSON-Unterstützung (SQL/JSON), Row Pattern Matching
SQL:2019	2019	Multimediale Erweiterungen (SQL/MM)
SQL:2023	2023	Property Graph Queries (SQL/PGQ), weitere JSON/CBOR-Erweiterungen

5.3 Was der Standard regelt

SQL ist in mehrere funktionale Bereiche unterteilt:

DDL – Data Definition Language — Datenstrukturen anlegen, ändern und löschen:

- `CREATE TABLE`, `ALTER TABLE`, `DROP TABLE`
- `CREATE INDEX`, `CREATE VIEW`, `CREATE SCHEMA`
- Constraints: `PRIMARY KEY`, `FOREIGN KEY`, `UNIQUE`, `CHECK`, `NOT NULL`

DML – Data Manipulation Language — Daten abfragen und verändern:

- `SELECT` mit `WHERE`, `GROUP BY`, `HAVING`, `ORDER BY`
- JOIN-Typen: `INNER JOIN`, `LEFT OUTER JOIN`, `RIGHT OUTER JOIN`, `FULL OUTER JOIN`, `CROSS JOIN`
- `INSERT`, `UPDATE`, `DELETE`, `MERGE` (seit SQL:2003)
- Common Table Expressions: `WITH`, `WITH RECURSIVE`
- Window Functions: `ROW_NUMBER()`, `RANK()`, `DENSE_RANK()`, `SUM() OVER (...)` u. a.

TCL – Transaction Control Language — Transaktionsgrenzen festlegen:

- `COMMIT`, `ROLLBACK`, `SAVEPOINT`

DCL – Data Control Language — Zugriffsrechte verwalten:

- `GRANT`, `REVOKE`

Weitere Spezifikationen umfassen: Cursor, Stored-Procedure-Sprache (SQL/PSM), Zeichensätze und Kollationen, Diagnoseschnittstellen sowie das Informationsschema (`INFORMATION_SCHEMA`).

5.4 Konformitätsstufen

SQL-92 führte drei formale Konformitätsstufen ein:

- **Entry SQL**: Kernfunktionalität — das, was jedes ernstzunehmende System beherrscht
- **Intermediate SQL**: Erweiterte Funktionen
- **Full SQL**: Vollständiger Standard

Kein verbreitetes Datenbanksystem implementiert den Standard vollständig. Das ist kein Versagen, sondern Realität: Manche Standardteile adressieren Szenarien, die bestimmte Systeme schlicht nicht anbieten müssen.

5.5 Standard und Praxis

In der Praxis gilt:

- Kernfunktionen (`SELECT`, `JOIN`, `GROUP BY`, `WHERE`, `COMMIT`, `ROLLBACK`) sind praktisch überall kompatibel.
- Fortgeschrittene Funktionen — Window Functions, temporale Abfragen, JSON-Operatoren — variieren in Syntax und Verhalten je nach System.
- Alle großen Systeme bieten proprietäre Erweiterungen, die über den Standard hinausgehen.

Wer migrationsfreundlichen Code schreiben will, priorisiert Standard-SQL im Kernteil und dokumentiert proprietäre Erweiterungen explizit.

5.6 Bedeutung für die Datenbankauswahl

SQL-Standardnähe ist ein legitimes Auswahlkriterium — besonders dann, wenn:

- Wissenstransfer zwischen Systemen wichtig ist,
- Tooling und ORM-Unterstützung breit verfügbar sein muss,
- ein zukünftiger Systemwechsel nicht ausgeschlossen ist.

Für Lernende gilt: Standard-SQL ist das tragfähige Fundament. Alle proprietären Erweiterungen sind Schichten darüber.

6. Wie werden Daten in Datenbanken gespeichert?

Jetzt kommt die physische Ebene. Sie ist der Grund, warum Datenbanken trotz Dateisystem so viel schneller und kontrollierter arbeiten als eine naive Dateiverarbeitung.

6.1 Vom Fachmodell zur physischen Struktur

Auf der Modellierungsebene gibt es zum Beispiel Entitäten wie `Kunde`, `Bestellung`, `Rechnung` oder `Sensorwert`. Das ist die fachliche Sicht.

Auf der physischen Ebene wird daraus etwas völlig anderes:

- Dateien,
- Segmente,
- Extents,
- Seiten,
- Zeilenformate,
- Indexseiten,
- Logblöcke,
- Freispeicherkarten,
- Hintergrundjobs.

Zwischen beiden Ebenen liegt also eine Übersetzungsschicht. Genau deshalb ist Datenbankmodellierung so wichtig: Ein schlechtes logisches Modell verursacht fast immer auch schlechte physische Zugriffswege.

6.2 Seiten statt "ganze Dateien"

Datenbanken arbeiten intern fast nie mit "ganzen Dateien" als elementarer Einheit, sondern mit Seiten oder Pages. Eine Seite ist ein fester Block, häufig im Bereich von einigen Kilobytes.

Die Datenbank denkt also nicht:

> "Lies bitte die komplette Datei `kunden.dat`."

Sondern eher:

> "Hole mir Seite 18472 der Tabelle und Seite 91 des passenden Index."

Das ist ein fundamentaler Unterschied.

Die Seitensicht bringt mehrere Vorteile:

- gezielte I/O statt Vollscan,
- einfache Verwaltung im Buffer Pool,
- effiziente Sperren und Latches,
- gute Wiederverwendung im Cache,
- klare Zuordnung von Datenblöcken zu Dateioffsets.

6.3 Typischer Aufbau einer Datenseite

Eine Datenseite enthält nicht einfach nur rohe Datensätze hintereinander. Typischerweise findet man:

- einen Seitenkopf mit Metadaten,
- einen Bereich für Zeilen oder Tupel,
- eine Slot-Tabelle oder Zeilenzeiger,
- Informationen über freien Speicher,
- Prüf- oder Verwaltungsinformationen.

Dadurch kann die Datenbank Zeilen verschieben, löschen oder erweitern, ohne dass jede Referenz im ganzen System ungültig wird.

6.4 Zeilenformat

In einer klassischen Row-Store-Datenbank enthält eine Zeile typischerweise:

- feste Spaltenwerte,
- variable Spaltenwerte,
- Null-Bitmaps,
- Versionsinformationen,
- Zeiger auf ausgelagerte LOB-Werte,
- interne Verwaltungsdaten.

Große Werte wie Texte, JSON-Dokumente, Bilder oder BLOBs werden häufig nicht vollständig inline in derselben Seite gehalten, sondern über Overflow- oder LOB-Strukturen ausgelagert.

6.5 Indexe sind eigene Datenstrukturen

Ein Index ist keine "Markierung" auf einer Tabelle, sondern meistens eine eigene physische Struktur. Sehr oft ist das ein B+-Baum.

Ein B+-Baum besteht vereinfacht aus:

- einer Root-Seite,
- mehreren inneren Seiten,
- Leaf-Seiten mit Schlüsseln und Verweisen auf Datensätze oder direkt auf die Daten.

Das Entscheidende ist: Ein B+-Baum ist flach. Selbst sehr große Tabellen haben oft nur wenige Ebenen. Dadurch muss eine Suche nicht Millionen Zeilen prüfen, sondern nur wenige Seiten entlang eines Pfades durchlaufen.

6.6 Heap, Cluster, Primary Key und Secondary Index

Nicht jede Datenbank speichert Tabellen gleich.

Mögliche Varianten:

- Heap-Organisation: Zeilen liegen unsortiert in Datenseiten.
- Clustered Storage: Die eigentlichen Daten sind nach dem Primärschlüssel oder einem Clustering-Schlüssel organisiert.
- Columnar Storage: Werte derselben Spalte liegen zusammen.

Das hat große Auswirkungen:

- auf Punktzugriffe,
- auf Bereichsabfragen,
- auf Schreibverhalten,
- auf Kompression,
- auf Wartungsaufwand.

6.7 Write-Ahead Logging

Eine der wichtigsten Ideen moderner Datenbanken lautet:

> Schreibe die Änderung zuerst in ein Log, bevor die eigentliche Datenseite endgültig auf Platte landet.

Dieses Prinzip nennt man Write-Ahead Logging.

Warum ist das so wichtig?

- Logschreiben ist oft sequentiell und damit sehr schnell.
- Bei einem Absturz kann das System anhand des Logs rekonstruieren, was bestätigt war und was nicht.
- Datenseiten müssen nicht bei jeder Änderung sofort synchron auf Platte geschrieben werden.

Die Datenbank gewinnt also zugleich Sicherheit und Geschwindigkeit.

6.8 Redo und Undo

Viele relationale Systeme trennen logisch zwischen:

- Redo-Information: Wie bestätigte Änderungen wiederhergestellt werden.
- Undo-Information: Wie unbestätigte oder ältere Versionen zurückgerollt oder sichtbar gemacht werden.

Gerade bei MVCC-Systemen und lang laufenden Transaktionen ist diese Trennung wichtig.

6.9 MVCC und Versionsverwaltung

Viele moderne Datenbanken arbeiten mit Multi-Version Concurrency Control. Dabei überschreibt eine Transaktion Daten nicht einfach blind für alle anderen sichtbar. Stattdessen existieren zeitweise mehrere Versionen derselben logischen Zeile.

Das bringt große Vorteile:

- Leser blockieren Schreiber seltener.
- Schreiber blockieren Leser seltener.
- Konsistente Schnappschüsse sind möglich.

Der Preis dafür:

- alte Versionen müssen verwaltet werden,
- Aufräumprozesse wie Vacuum, Purge oder Garbage Collection werden wichtig,
- Speicher- und I/O-Verhalten wird komplexer.

6.10 LSM-Trees, SSTables und Compaction

Nicht jede Datenbank aktualisiert Datenseiten direkt an Ort und Stelle. Viele verteilte oder write-optimierte Systeme arbeiten mit LSM-ähnlichen Speicheransätzen.

Typischer Ablauf:

1. Ein Write landet zunächst im Commit Log.
2. Gleichzeitig wird er in einer In-Memory-Struktur wie einer Memtable eingetragen.
3. Später wird diese Struktur als unveränderliche Datei auf Platte geschrieben.
4. Mehrere solcher Dateien werden im Hintergrund zusammengeführt und bereinigt.

Diese unveränderlichen Dateien heißen oft SSTables oder ähnlich.

Der große Vorteil:

- hohe Schreibgeschwindigkeit,
- viele sequentielle Schreibvorgänge,
- gute Eignung für verteilte Systeme.

Die Kehrseite:

- Reads müssen eventuell mehrere Dateien berücksichtigen,
- Compaction kostet Ressourcen,
- Read Amplification und Write Amplification müssen sauber gemanagt werden.

Für genau diese Welt ist Apache Cassandra ein sehr gutes Beispiel.

6.11 Spaltenorientierte Speicherung

In einem Column Store werden Werte derselben Spalte zusammenhängend gespeichert. Das ist ideal für analytische Abfragen wie:

```
SELECT land, SUM(umsatz)
FROM fakten
WHERE jahr = 2025
GROUP BY land;
```

Warum? Weil die Datenbank nicht jede vollständige Zeile lesen muss, sondern primär nur die Spalten `land`, `umsatz` und `jahr`.

Das reduziert I/O drastisch und verbessert Kompression, etwa durch:

- Dictionary Encoding,
- Run-Length Encoding,
- Delta Encoding,
- Bit-Packing.

Ein wichtiges Beispiel für diese Denkrichtung ist SAP HANA / HANADB.

6.11.1 Row Store vs. Column Store: Entscheidungssicht

Die Frage "Row Store oder Column Store?" ist keine rein technische Detailfrage, sondern eine der wichtigsten Architekturentscheidungen für den späteren Workload.

Kriterium	Row Store	Column Store
Stark bei	Einzeltransaktionen, Punktzugriffe, häufige kleine Updates	Scans, Aggregationen, analytische Auswertungen
Physische Nähe	alle Spalten einer Zeile liegen zusammen	Werte derselben Spalte liegen zusammen
Typischer Workload	OLTP	OLAP
Schreibverhalten	sehr gut für häufige Zeilenänderungen	schwächer bei vielen kleinen Updates

Kriterium	Row Store	Column Store
Kompression	begrenzt	oft sehr stark
Typische Folgefrage	gute Indizes und MVCC	gute Sortierung, Encoding, Scan-Optimierung

Faustregel: Wenn sehr häufig ganze Geschäftsobjekte gelesen und geschrieben werden, ist ein Row Store meist näher an der operativen Realität. Wenn große Datenmengen spaltenweise gefiltert, gruppiert und aggregiert werden, spielt ein Column Store seine Stärken aus.

6.12 Kataloge und Metadaten

Die Datenbank speichert nicht nur Nutzdaten. Sie speichert auch Metadaten:

- welche Tabellen existieren,
- welche Spalten sie haben,
- welche Datentypen gelten,
- welche Indizes vorhanden sind,
- welche Statistiken aktuell sind,
- welche Benutzer welche Rechte besitzen.

Ohne diesen Systemkatalog könnte der Optimizer gar nicht arbeiten.

7. Warum ist der Zugriff auf Datenbanken so schnell, obwohl die Daten im Filesystem liegen?

Das ist eine der wichtigsten Fragen überhaupt. Die Antwort lautet:

> Datenbanken sind schnell, weil sie gerade nicht wie ein allgemeines Dateisystem auf Daten zugreifen.

Sie benutzen das Dateisystem nur als unterste Persistenzschicht. Alles darüber ist hoch spezialisiert.

7.1 Das Dateisystem ist die Basis, nicht die eigentliche Logik

Ja, die Datenbankdateien liegen letztlich im Dateisystem. Das Dateisystem liefert:

- Blöcke,
- Dateihandles,
- Pufferung auf Betriebssystemebene,
- Rechte,
- Persistenz auf SSD oder HDD.

Aber das Dateisystem weiß nicht:

- was eine Zeile ist,
- was ein Index ist,
- welche Transaktion sichtbar ist,
- wo freie Slots liegen,
- welche Seiten heiß sind,
- welche Abfrage als Nächstes wahrscheinlich kommt.

Diese Intelligenz liefert die Datenbank selbst.

7.2 Indizes verkleinern den Suchraum radikal

Der wichtigste Geschwindigkeitshebel ist nicht "schnellere Platte", sondern "weniger lesen".

Wenn du ohne Index in 100 Millionen Zeilen nach einer Kundennummer suchst, bleibt oft nur ein großer Scan.

Mit einem B+-Baum-Index passiert etwas völlig anderes:

1. Root-Seite lesen oder aus dem Cache nehmen.
2. Über innere Seiten zum relevanten Schlüsselbereich navigieren.
3. Leaf-Seite lesen.
4. Datenseite laden oder direkt den Treffer aus dem Index liefern.

Statt Millionen Zeilen zu prüfen, werden nur wenige Seiten gelesen. Genau deshalb können Datenbanken auch auf sehr großen Datenmengen schnell sein.

7.3 Der Buffer Pool hält heiße Daten im RAM

Der zweite große Geschwindigkeitshebel ist Caching.

Sehr grob gilt:

- RAM ist viel schneller als SSD.
- SSD ist viel schneller als HDD.
- Wiederverwendung im RAM ist schneller als jede erneute I/O.

Der Buffer Pool oder Buffer Cache hält daher:

- häufig genutzte Datenseiten,
- Wurzeln und obere Ebenen von Indizes,
- Systemkatalogseiten,
- oft auch temporäre Arbeitsbereiche.

In realen Systemen liegen Root- und viele innere Indexseiten fast dauerhaft im Speicher. Dadurch kostet der erste Teil einer Suche praktisch keine physische I/O mehr.

7.4 Seitenlokalität und Read-Ahead

Datenbanken versuchen, Daten so zu organisieren, dass räumliche Nähe ausgenutzt wird:

- zusammengehörige Schlüsselbereiche in benachbarten Seiten,
- Clusterung nach Primärschlüssel,
- Segmentierung nach Partitionen,
- Vorab-Lesen zusammenhängender Seiten.

Wenn das System erkennt, dass eine Bereichsabfrage läuft, kann es weitere Seiten vorsorglich laden. Dieses Read-Ahead vermeidet Wartezeiten.

7.5 Logs machen Schreiben schneller

Viele Anfänger erwarten, dass jede Änderung sofort die eigentliche Datenseite synchron auf Platte schreibt. Das wäre langsam.

Stattdessen passiert meist:

1. Die Änderung wird im Log protokolliert.
2. Die Datenseite wird im Speicher geändert.
3. Später schreibt ein Hintergrundprozess die Datenseite kontrolliert weg.

Das ist oft schneller, weil Logschreiben sequentiell erfolgt. Sequentielle I/O ist für Speichermedien und Controller leichter effizient zu bedienen als viele kleine zufällige Schreibzugriffe.

7.6 Der Optimizer vermeidet unnötige Arbeit

Schnelligkeit bedeutet nicht nur schnelle I/O, sondern auch gute Ausführungspläne.

Der Optimizer nutzt Statistiken über:

- Tabellenvolumina,
- Werteverteilungen,
- Kardinalitäten,
- Selektivität von Filtern,
- Indexverfügbarkeit.

Dadurch kann er die voraussichtlich günstigste Ausführungsstrategie wählen. Gute Statistiken sind deshalb ein Performance-Thema, nicht bloß Verwaltung.

7.7 Nebenläufigkeit wird kontrolliert, nicht dem Zufall überlassen

Wenn Hunderte oder Tausende Nutzer gleichzeitig arbeiten, reicht ein schneller Dateizugriff nicht aus. Entscheidend ist, dass sich parallele Prozesse nicht chaotisch gegenseitig blockieren.

Datenbanken setzen dafür auf:

- Locks,
- Latches,
- MVCC,
- Version Chains,
- Deadlock Detection,
- Isolation Levels.

Diese Mechanismen kosten zwar selbst etwas Rechenzeit, verhindern aber, dass das System in inkonsistente oder unkontrollierbar langsame Zustände kippt.

7.8 Moderne Speicherhardware wird gezielt genutzt

Datenbanken profitieren stark von:

- SSD und NVMe,
- großen RAM-Kapazitäten,
- mehreren CPU-Kernen,
- asynchroner I/O,
- paralleler Ausführung,
- NUMA-bewusstem Speicherzugriff in High-End-Systemen.

Aber Hardware allein ersetzt kein gutes Datenmodell. Schlechte Indizes, schlechte Partitionierung oder falsche Zugriffspfade können auch auf teurer Hardware langsam bleiben.

7.9 Kompression erhöht oft die Geschwindigkeit

Kompression klingt zunächst nach Zusatzaufwand. In Datenbanken führt sie aber oft zu mehr Tempo, weil weniger Daten von Platte oder durch CPU-Caches transportiert werden müssen.

Wenn das System 20 MB komprimierte Daten liest statt 120 MB unkomprimierte Daten, ist die kleine Dekompressionsarbeit häufig günstiger als die gesparte I/O.

7.10 Ein gedankliches Beispiel

Nehmen wir eine große Kundentabelle mit 50 Millionen Zeilen und einer Suche nach `kundennummer = 4711`.

Ohne Index:

- viele oder alle Datenseiten lesen,
- jede Zeile prüfen,
- hohe I/O-Last,
- schlechte Latenz.

Mit gutem Index und warmem Cache:

- Root-Seite bereits im RAM,
- innere Indexseiten bereits im RAM,
- Leaf-Seite lesen,
- eine oder wenige Datenseiten lesen,
- Sichtbarkeit prüfen,
- Ergebnis liefern.

Das ist der Unterschied zwischen Dateidurchwühlen und datenbankgestütztem Zugriff.

8. B-Tree, Hash, Column Store und LSM: Warum die Speicherstruktur so viel ausmacht

Nicht nur die Frage "welche Datenbank", sondern auch "welche interne Struktur" entscheidet über Performance.

8.1 Partitionierung und Sharding

Sobald Datenmengen und Last stark wachsen, reicht die Wahl der internen Datenstruktur allein nicht mehr aus. Dann stellt sich die Frage, wie Daten horizontal verteilt werden.

- **Partitionierung** teilt große Tabellen oder Datenmengen in logisch getrennte Bereiche auf, zum Beispiel nach Datum, Region oder Mandant.
- **Sharding** verteilt Daten physisch auf mehrere Knoten, oft anhand eines Schlüssels wie Kundennummer oder Tenant-ID.

Beides beeinflusst massiv:

- die Skalierbarkeit,
- die Join-Fähigkeit über Knoten hinweg,
- Hotspots bei Schreiblast,
- Rebalancing und Betriebsaufwand,
- die Komplexität von Transaktionen und Konsistenz.

Ein System kann also an der Speicherstruktur stark sein und dennoch operativ scheitern, wenn Partitionierung oder Sharding schlecht gewählt sind.

Struktur	Stark bei	Schwächer bei
B+-Baum	Punktzugriffe, Bereichsabfragen, sortierte Zugriffe	sehr write-lastige LSM-Szenarien können günstiger sein
Hash-Struktur	exakte Gleichheitssuchen	Bereichsabfragen, Sortierung
LSM-Ansatz	sehr hohe Write-Last, verteilte Systeme	Compaction-Aufwand, mehrere Read-Pfade
Column Store	Aggregation, Scans, analytische Queries	häufige kleine Zeilenupdates

Darum ist es so wichtig, die Datenbank nicht nur nach Markennamen, sondern nach innerem Arbeitsmodell zu wählen.

9. Transaktionen, Konsistenz und Recovery

Eine Datenbank ist nicht nur schnell, sondern im Idealfall auch korrekt.

9.1 ACID

ACID ist kein Marketing-Schlagwort, sondern beschreibt die Mindestfrage:

- Wird eine Änderung vollständig oder gar nicht wirksam?
- Bleibt das System in einem gültigen Zustand?
- Beeinflussen sich parallele Vorgänge korrekt?
- Überlebt ein bestätigter Commit einen Absturz?

9.1.1 Isolation Levels

Gerade das I in ACID wird in der Praxis oft zu grob verstanden. Isolation bedeutet nicht einfach nur "parallel, aber korrekt", sondern die bewusste Wahl eines Isolationsniveaus mit klaren Nebenwirkungen.

Isolation Level	Verhindert typischerweise	Typische Kehrseite
Read Uncommitted	fast nichts	unsaubere Reads möglich
Read Committed	Dirty Reads	Non-Repeatable Reads bleiben möglich
Repeatable Read	Dirty Reads und viele Non-Repeatable Reads	Phantom Reads oder höhere Sperrkosten je nach System
Serializable	Anomalien weitgehend vollständig	höchste Kosten, geringere Parallelität

In realen Systemen ist die Wahl des Isolation Levels ein Performance- und Korrektheitsthema zugleich. Höhere Isolation erhöht meist die Sicherheit, kann aber Parallelität, Latenz und Durchsatz verschlechtern.

9.2 Crash Recovery

Wenn ein Server mitten im Schreiben abstürzt, darf die Datenbank nicht in einen undefinierten Zustand fallen. Genau dafür dienen:

- Logs,
- Checkpoints,
- Recovery-Algorithmen,
- Undo- und Redo-Phasen,
- Prüfsummen und Konsistenzmechanismen.

9.3 Replikation

Viele Datenbanken kopieren Änderungen auf weitere Knoten. Ziele sind:

- höhere Verfügbarkeit,
- Failover,
- Lastverteilung,
- Disaster Recovery,
- geografische Nähe.

Aber Replikation ist kein Gratis-Feature. Sie wirft Fragen auf:

- synchron oder asynchron,
- ein primärer Knoten oder mehrere Schreibknoten,
- welches Konsistenzniveau ist fachlich akzeptabel,
- welche Latenz darf entstehen.

9.4 CAP-Theorem und BASE

Für verteilte Systeme ist das CAP-Theorem fundamental. Es besagt, dass ein verteiltes System gleichzeitig nur zwei der drei folgenden Eigenschaften garantieren kann:

- **Consistency (Konsistenz)**: Alle Knoten liefern denselben aktuellen Zustand.
- **Availability (Verfügbarkeit)**: Jede Anfrage erhält eine Antwort.
- **Partition Tolerance (Partitionstoleranz)**: Das System funktioniert trotz Netzwerkausfällen zwischen Knoten.

Da Partitionstoleranz in verteilten Systemen praktisch nicht verzichtbar ist, stehen Architekten vor der Entscheidung zwischen CP-Systemen (Konsistenz + Partitionstoleranz) und AP-Systemen (Verfügbarkeit + Partitionstoleranz).

Viele NoSQL-Systeme verzichten bewusst auf starke Konsistenz und wählen ein Konsistenzmodell, das als BASE beschrieben wird:

- **Basically Available**: Das System antwortet auf Anfragen, auch wenn nicht alle Knoten synchron sind.
- **Soft State**: Der Zustand des Systems kann sich ohne externe Eingabe ändern, weil Synchronisation nachgelagert erfolgt.
- **Eventually Consistent**: Über Zeit werden alle Knoten denselben Zustand erreichen — aber nicht notwendigerweise sofort.

BASE steht damit im Gegensatz zu ACID. Weder ist ACID falsch noch BASE – sie stellen unterschiedliche Abwägungen für unterschiedliche Workloads dar.

9.5 Backup, Restore und Point-in-Time Recovery

Replikation wird oft mit Datensicherung verwechselt. Das ist ein gefährlicher Denkfehler.

- **Replikation** schützt primär vor Knotenausfällen und verbessert Verfügbarkeit.
- **Backup** schützt vor Datenverlust durch Bedienfehler, Softwarefehler, logische Korruption oder Ransomware.
- **Point-in-Time Recovery (PITR)** erlaubt die Wiederherstellung auf einen exakt definierten Zeitpunkt, etwa direkt vor einer fehlerhaften Massenlöschung.

Ein replizierter Fehler bleibt ein Fehler: Wenn eine falsche Löschung oder beschädigte Änderung repliziert wird, ist sie auf mehreren Knoten gleichzeitig vorhanden. Genau deshalb gehören Backup-Strategie, Restore-Tests und Aufbewahrungsfristen zu jeder ernsthaften Datenbankarchitektur.

10. Datenqualität, ETL und Datenbanken

Eine Datenbank kann Daten sehr zuverlässig speichern. Sie garantiert aber nicht automatisch, dass die Daten fachlich gut sind.

Hier kommen zwei Themen ins Spiel:

1. Datenqualität,
2. Datenintegration.

Wenn Daten aus vielen Systemen zusammenfließen, müssen sie extrahiert, transformiert und geladen werden. Genau dafür ist der ETL-Gedanke zentral. Mehr dazu findest du im Artikel Datenintegration – ETL.

Ebenso wichtig ist die Qualität der Daten vor Analyse oder maschinellem Lernen. Fehlerhafte, doppelte oder widersprüchliche Daten schaden nicht nur Modellen, sondern auch klassischen Auswertungen und Reports. Dazu passt die Vertiefung Datenbereinigung und seine Bedeutung für maschinelles Lernen.

Kurz gesagt:

- Eine gute Datenbank ersetzt keine gute Datenpflege.
- Ein schneller Query ersetzt keine saubere Semantik.
- Ein starkes Storage Engine Design ersetzt keine gute Datenintegration.

11. Cloud-Native und die Trennung von Compute und Storage

Lange Zeit waren Rechenleistung (Compute) und Datenspeicher (Storage) in Datenbanken untrennbar miteinander auf denselben physischen oder virtuellen Knoten verknüpft (Shared-Nothing-Architektur in der Skalierung). War die Festplatte voll, musste ein kompletter, teurer Server hinzugefügt werden — selbst wenn genügend Rechenleistung da war.

Moderne Cloud-native Datenbankarchitekturen brechen diese Monolithik auf, indem sie den Storage-Layer konsequent von der Compute-Ebene trennen (Separation of Compute and Storage).

- **Storage:** Die Daten landen in einem verteilten, extrem haltbaren und hochverfügbaren Logging- und Speichersystem (z. B. adaptierter Objektspeicher).
- **Compute:** Die tatsächlichen SQL-Ausführungsknoten sind zustandslos (stateless). Ihr Cache wird dynamisch gefüllt, aber die primäre Halte-Verantwortung liegt nicht lokal.

Dieser Paradigmenwechsel erlaubt es einer Analysedatenbank (etwa **Snowflake**), unbegrenzt Gigabytes abzuspeichern, ohne dass ständig teure CPUs laufen müssen. Erst für komplexe Abfragen werden die virtuellen Warehouses (Rechenknoten) dynamisch eingeschaltet und am Ende wieder schlafen gelegt. Genauso ist **Amazon Aurora** für relationale Datenbanken auf diesem Prinzip des disaggregierten Storage aufgebaut: Log-Datensätze werden an hunderte Speicherzellen im Hintergrund gestreamt, während Leser-Knoten nahezu sofort horizontal skaliert werden können.

Dieser architektonische Sprung ist heute die Definition echter Cloud-nativer Datenbanken, im Gegensatz zu bloß in VMs laufenden klassischen Datenbankinstallationen ('Lift and Shift').

11.1 Lakehouse und Warehouse in Microsoft Fabric

Moderne Datenplattformen trennen nicht nur Compute und Storage, sondern auch unterschiedliche Nutzungsmodelle für analytische Daten. Microsoft Fabric ist dafür ein gutes Beispiel, weil dort mehrere Datenparadigmen in einer Plattform zusammengeführt werden.

- **Lakehouse**: verbindet Data-Lake-Denken mit tabellarischer Auswertbarkeit. Dateien, offene Formate und tabellenartige Strukturen leben in einer gemeinsamen Umgebung. Das ist stark für Data Engineering, Data Science und flexible Exploration großer Rohdatenbestände.
- **Warehouse**: fokussiert stärker auf klassische relationale Analytik, SQL-zentrierte Modellierung, BI-Workloads und kontrollierte Abfrageschichten für Reporting und Self-Service Analytics.

Vereinfacht gesagt:

- Das **Lakehouse** ist näher an Rohdaten, Pipelines und offener Datenhaltung.
- Das **Warehouse** ist näher an kuratierten Modellen, Governance und klassischer analytischer SQL-Nutzung.

Für den allgemeinen Datenbanküberblick ist das wichtig, weil moderne Plattformen heute nicht mehr nur zwischen "Datenbank" und "Dateisystem" unterscheiden. Sie kombinieren Speicher, Tabellen, Pipelines, semantische Modelle und Analyse-Engines in einer gemeinsamen Architektur.

11.2 Governance, Lineage und Datensicherheit

Mit modernen Datenplattformen wächst nicht nur die technische Leistungsfähigkeit, sondern auch die Governance-Anforderung. Je mehr Teams, Pipelines, Berichte und Modelle auf denselben Datenraum zugreifen, desto wichtiger wird die Frage, wer was sehen, verändern und nachvollziehen darf.

Wichtige Begriffe sind:

- **Governance**: Regeln, Rollen und Prozesse dafür, wie Daten genutzt, geschützt und verantwortet werden.
- **Lineage**: Nachvollziehbarkeit, aus welchen Quellen eine Tabelle, Kennzahl oder ein Bericht entstanden ist.
- **Datensicherheit**: Zugriffsschutz, Verschlüsselung, Maskierung sensibler Daten und kontrollierte Berechtigungen.

Gerade in Plattformen wie Microsoft Fabric, klassischen Warehouses oder großen Lakehouse-Umgebungen ist das kein Randthema. Ohne Governance entsteht schnell ein technisch mächtiges, aber fachlich riskantes System: Berichte widersprechen einander, sensible Daten werden zu breit freigegeben und Änderungen sind kaum noch sauber rückverfolgbar.

Deshalb gehören zu einer reifen Datenarchitektur nicht nur Storage, Compute und SQL, sondern auch:

- rollenbasierte Berechtigungen,
- nachvollziehbare Datenherkunft,
- kuratierte Schichten zwischen Rohdaten und Reporting,
- Schutzmechanismen für sensible oder regulatorisch relevante Informationen.

12. Lernen, Testen und Prototyping

Wer Datenbanken ernsthaft verstehen will, sollte nicht nur Theorie lesen, sondern mit echten Datenmodellen und Beispielbeständen arbeiten. Dafür sind Demodatenbanken ideal, weil sie:

- typische Geschäftsobjekte enthalten,
- realistischere Abfragen erlauben,
- Join- und Analysefälle greifbar machen,
- Performance- und Modellierungsübungen ermöglichen.

Eine gute Einstiegssammlung dazu findest du bei den Beispieldatenbanken.

13. Wichtige Datenbanksysteme im Überblick

Dieser Abschnitt ist bewusst kein Ranking, sondern eine Einordnung nach Stärken und typischem Einsatzprofil.

System	Typische Einordnung	Typische Stärke	Interne Denkrichtung	Link
PostgreSQL	objektrelationale Datenbank	Standards, Erweiterbarkeit, Integrität	klassisches relationales Kernsystem mit viel Erweiterbarkeit	PostgreSQL
MySQL	relationale Datenbank	Web-Anwendungen, breite Verbreitung, einfacher Einstieg	relational, oft InnoDB-zentriert	MySQL
Microsoft SQL Server	relationale Datenbank	Microsoft-Ökosystem, BI, Enterprise-Workloads	relational mit starkem Tooling und Verwaltungsumfeld	MSSQL / Microsoft SQL Server
Oracle Database	relationale Datenbank	Enterprise-Funktionen, Hochverfügbarkeit, große Kernsysteme	relational mit starkem Fokus auf Skalierung, HA und Governance	Oracle Database
MongoDB	Dokumentendatenbank	flexible Schemata, schnelle Produktentwicklung	dokumentenorientiert, JSON/BSON-nah	MongoDB
Apache Cassandra	Wide-Column / verteilte NoSQL-DB	hohe Write-Last, horizontale Skalierung, Verfügbarkeit	LSM-basiert, partitioniert, verteilte Konsistenzprofile	Apache Cassandra
SAP HANA / HANADB	in-memory, spaltenorientiert, relational	analytische Geschwindigkeit, Kompression, Echtzeit-Szenarien	columnar, in-memory, hybrid aus operativem und analytischem Denken	SAP HANA / HANADB
Microsoft Fabric Lakehouse	Lakehouse / analytische Plattform	Rohdaten, offene Tabellenformate, Data Engineering und Analytics in einer Plattform	lakehouse-orientiert, cloud-nativ, plattformintegriert	https://learn.microsoft.com/fabric/data-engineering/lakehouse-overview

System	Typische Einordnung	Typische Stärke	Interne Denkrichtung	Link
Microsoft Fabric Warehouse	analytisches Warehouse	SQL-zentrierte BI- und Reporting-Workloads in Fabric	warehouse-orientiert, cloud-nativ, analytisch	https://learn.microsoft.com/fabric/data-warehouse/data-warehousing

14. Wie wählt man die richtige Datenbank aus?

Die beste Entscheidung entsteht selten aus Markenpräferenz. Sie entsteht aus einer guten Fragenliste.

14.1 Fragen zum Workload

- Sind die Zugriffe transaktional oder analytisch?
- Gibt es viele kleine Writes oder wenige große Bulk Loads?
- Überwiegen Punktzugriffe, Bereichsabfragen, Aggregationen oder Graph-Traversals?
- Wie stark schwankt die Last?

14.2 Fragen zum Datenmodell

- Sind die Beziehungen streng und stabil?
- Ändert sich das Schema häufig?
- Sind Dokumente natürlich verschachtelt?
- Müssen komplexe Joins regelmäßig performant sein?

14.3 Fragen zum Betrieb

- Wird horizontale Skalierung benötigt?
- Welche Verfügbarkeit wird verlangt?
- Ist Multi-Region-Betrieb nötig?
- Wie viel Betriebs-Know-how ist im Team vorhanden?

14.4 Fragen zu Governance und Compliance

- Welche Rechte- und Audit-Anforderungen bestehen?
- Gibt es Anforderungen an Verschlüsselung, Maskierung oder Mandantentrennung?
- **RPO (Recovery Point Objective):** Wie viel Datenverlust ist bei einem Server- und Leitungsabsturz maximal hinnehmbar (z. B. die Transaktionen der letzten 5 Minuten oder strikt null)?
- **RTO (Recovery Time Objective):** Wie schnell muss das Gesamtsystem nach einem katastrophalen Ausfall operativ wieder verfügbar sein (z.B. in 30 Sekunden durch eine automatische Failover-Replik)?

15. Häufige Missverständnisse über Datenbanken

15.1 "NoSQL ist immer schneller"

Nein. NoSQL-Systeme können in bestimmten Lastprofilen deutlich besser passen. Für viele geschäftskritische Workloads mit Integritätsregeln, Joins und Transaktionen ist eine relationale Datenbank aber oft die bessere Wahl.

15.2 "Wenn alles im RAM ist, ist jedes Problem gelöst"

Auch In-Memory-Systeme brauchen:

- gute Indizes,
- saubere Modellierung,

- kontrollierte Parallelität,
- Recovery,
- durchdachte Persistenz.

RAM ersetzt keine Architektur.

15.3 "Die Daten liegen sowieso in Dateien, also kann ich das auch selbst bauen"

Man kann sehr vieles "selbst bauen". Der schwierige Teil ist nicht das Wegschreiben von Bytes, sondern:

- Crash Recovery,
- konsistente Parallelität,
- Optimizer-Logik,
- Indizes,
- Checkpointing,
- Statistiken,
- Replikation,
- Rechteverwaltung,
- Betrieb über Jahre.

Genau deshalb sind Datenbanksysteme so komplex und so wertvoll.

16. Fazit

Eine Datenbank ist ein hochspezialisiertes System zur Verwaltung persistenter Daten. Sie ist schnell, weil sie Daten nicht blind als Dateien behandelt, sondern logisch und physisch organisiert: mit Seiten, Caches, Indizes, Logstrukturen, Optimizer-Entscheidungen und kontrollierter Nebenläufigkeit.

Wer Datenbanken wirklich verstehen will, sollte immer diese Leitfrage mitdenken:

> Welche fachliche Anfrage wird durch welche logische Struktur und welche physische Speichermechanik beantwortet?

Sobald diese Verbindung klar ist, werden viele Dinge plötzlich verständlich:

- warum gute Modellierung so wichtig ist,
- warum Indizes oft mehr bringen als rohe Hardware,
- warum Logs für Performance und Sicherheit zugleich relevant sind,
- warum unterschiedliche Datenbankarten in unterschiedlichen Welten stark sind,
- und warum das Dateisystem nur die Grundlage ist, nicht aber die eigentliche Intelligenz.

Über den Autor

Erhard Rainer berät kleine und mittlere Unternehmen in Österreich zu Business Intelligence, Machine Learning und KI-Agenten. Er hat 15 Jahre Erfahrung in Data Analytics, davon 10 Jahre in der Leitung von Teams und Departments in Konzernen, und ist Dozent für betriebswirtschaftliche und Data-Analytics-Fächer.

Sprechen Sie mit mir über Ihre Daten: Das Erstgespräch ist kostenlos.

https://www.thinkbigger.at/contact.html#anfrage

© 2026 Think BIGGER – Erhard Rainer. Dieses PDF gibt den Stand des Online-Artikels vom 10.04.2026 wieder; maßgeblich ist die Online-Fassung unter https://www.thinkbigger.at/wissen/datenbanken/