DATENBANKEN I
DIE AUSWAHL VON DATENBANKEN

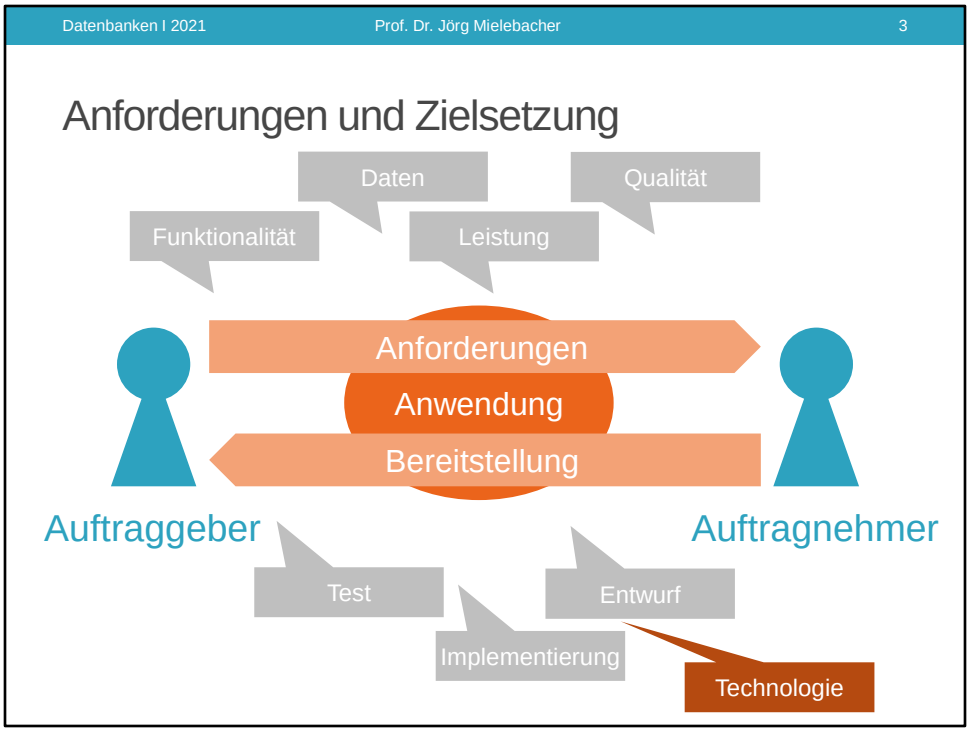
Prof. Dr. Jörg Mielebacher
joerg.mielebacher@mosbach.dhbw.de

Die folgenden Folien beschäftigen sich mit der Auswahl von Datenbanken. Zu jeder Folie sind Notizenseiten erfasst.

Verbesserungsvorschläge und Fehlerhinweise können Sie gerne an die Adresse joerg.mielebacher@mosbach.dhbw.de senden.

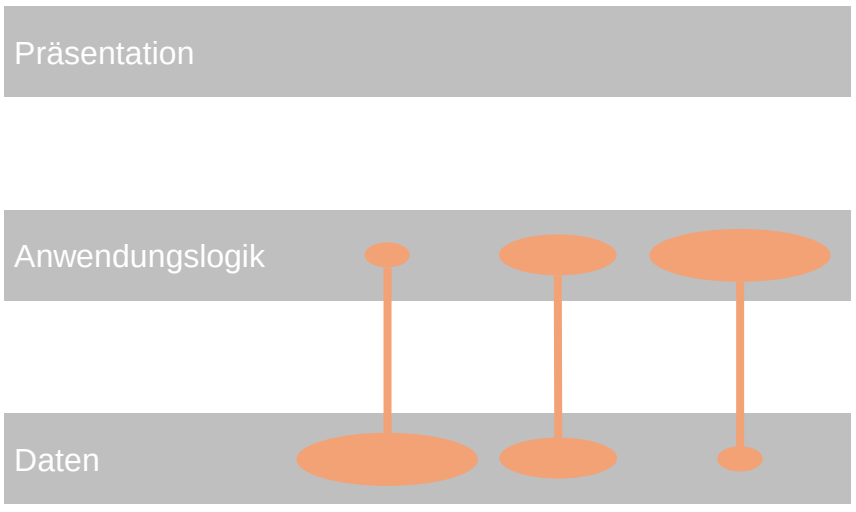
Grundsatzfragen

Die Auswahl von Datenbanken fügt sich in den sehr viel größeren Kontext der Software-Entwicklung ein. Es geht also nicht darum, dasjenige DBMS einzusetzen, das gerade in Mode ist, sondern dasjenige, das die bestehenden Anforderungen am besten erfüllt.

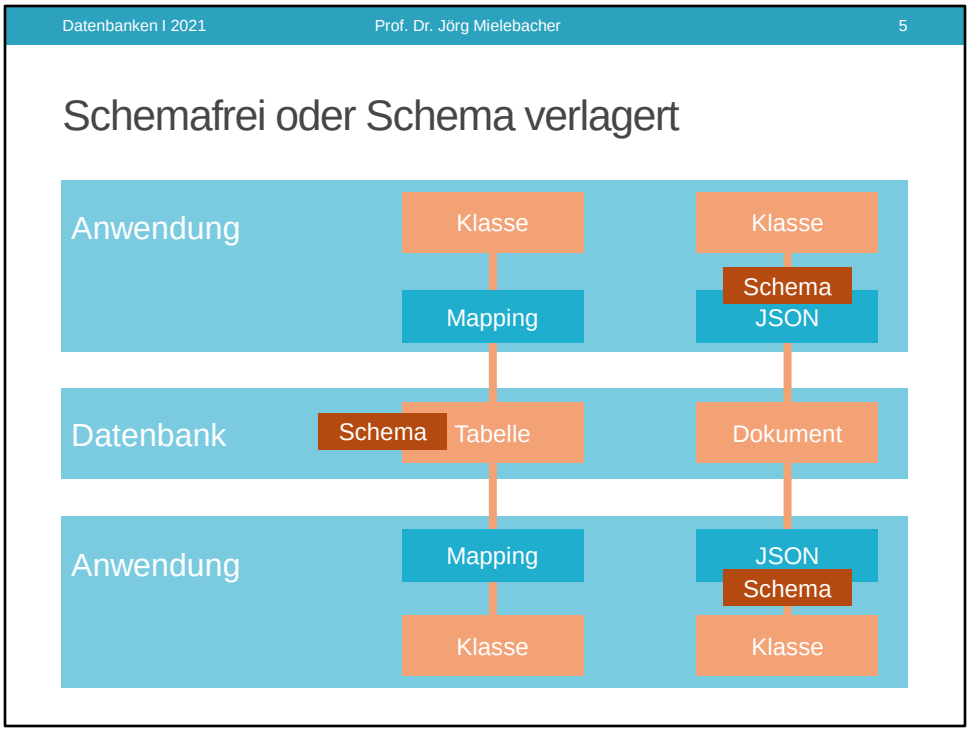


Hier hilft es, wenn man sich das Grundprinzip der Software-Entwicklung in Erinnerung ruft: Die Anforderungen des Auftraggebers erlauben es dem Auftragnehmer, eine geeignete Lösung bereitzustellen, die das Problem des Auftraggebers löst und ihm entsprechenden Nutzen stiftet. Wichtig für die Auswahl der Datenbank sind neben der Funktionalität vor allem Anforderungen hinsichtlich der Daten, der Leistung und der Qualität (z.B. Sicherheit, Wartbarkeit usw.). Die Bereitstellung wiederum umfasst Entwurf, Implementierung und Test, wobei die eigentliche Technologieauswahl (hier: die Wahl des DBMS) üblicherweise in Entwurf oder Implementierung erfolgt. Tests wiederum können helfen, diese Entscheidung zu untermauern, da bei vielen DBMS die Leistung stark von den verwendeten Daten, der Hardware und der Art der Abfragen abhängt.

Anwendung und Datenbank



Hinsichtlich des Anwendungsentwurfs gibt es ebenfalls wichtige Grundsatzfragen: Legt man beispielsweise eine 3-Schicht-Architektur zugrunde, so stellt sich die Frage, wo und in welchem Umfang Anwendungslogik angesiedelt ist. Früher hat man häufig große Teile der Anwendungslogik in der Datenbank abgebildet (z.B. durch Trigger, Constraints und Stored Routines). Heute gibt es unter Experten intensive Diskussionen darüber, ob dies noch zeitgemäß ist. So wird häufig angeführt, dass Anwendungslogik in der Datenbankschicht deutlich schlechter skalierbar ist, als in der Anwendungslogikschicht, wo sie von (gut skalierbaren) Applikationsservern bereitgestellt wird. Andererseits ist davon auszugehen, dass gerade das DBMS sehr viel effizienter komplexe Abfragen ausführen kann, die man sonst aufwändig in der Anwendung selbst durchführen müsste (Filterung, Joins usw.). Die Umsetzung von Anwendungslogik in der Datenbank kann außerdem problematisch sein, wenn es zu Code-Duplikation kommt oder wenn solche Anwendungslogik nicht ausreichend in Tests bzw. Build-Management eingebunden sind.



NoSQL-Datenbanken werden häufig mit der sog. Schemafreiheit motiviert. In relationalen Datenbanken liegt ein Datenbankschema zugrunde, durch das der Aufbau jeder Tabelle vollständig und eindeutig festgelegt ist. Klassen der Anwendung werden auf diesen Aufbau der Tabellen abgebildet, also z.B. die Zuordnung von Klassenattributen zu Tabellenspalten. Ein Objekt der Klasse kann dann als Datensatz in der Tabelle abgelegt werden.

Verwendet man stattdessen eine dokumentenorientierte Datenbank, so wird das Objekt z.B. in ein JSON-Dokument überführt, das dann wiederum in der Datenbank abgelegt wird – die Datenbank ist in diesem Fall schemafrei, allerdings ist für das JSON-Dokument dennoch ein Schema notwendig, wenn man es einheitlich verwenden möchte; Attribute müssen einheitlich benannt sein, die Struktur muss verbindlich festgelegt sein usw. Ist dies nicht erfüllt, kann es bei der Nutzung der Dokumente zu Fehlern kommen. Das Schema verlagert sich also von der Datenbank in die Anwendung.

Der Unterschied zwischen Schema in der Datenbank und in der Anwendung wird besonders deutlich, wenn man als Szenario die Integration anhand einer Datenbank annimmt, d.h. eine Anwendung schreibt in die Datenbank, eine andere Anwendung liest aus der Datenbank – gleiches gilt natürlich auch für Komponenten und dergleichen. In diesem Fall müssen beide Partner dasselbe Schema der gespeicherten Dokumenten zugrunde legen können.

No SQL, Not only SQL, Not yet SQL

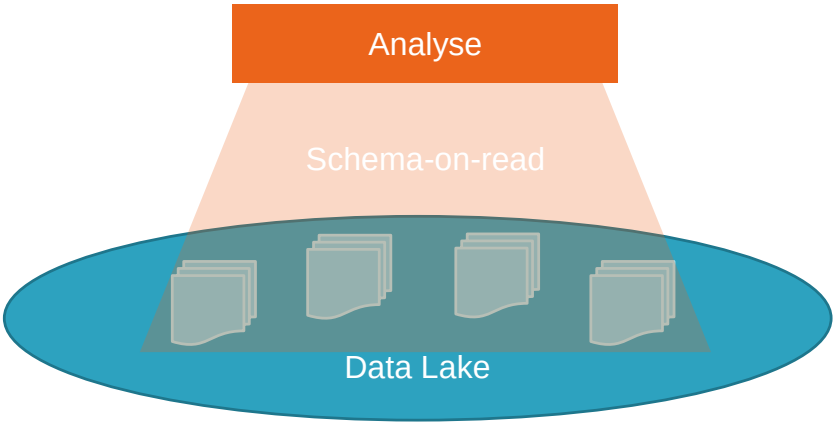
My favorite way to categorize the NoSQL guys is that they started off as "NoSQL," meaning "SQL is bad." After a while, that turned into NoSQL meaning "not only SQL" – SQL was fine, and they wanted to co-exist with SQL systems. My prediction is that NoSQL will come to mean "not yet SQL".

Michael Stonebraker

In der Diskussion über NoSQL-Datenbanken hat der Datenbank-Pionier Michael Stonebraker 2013 eine interessante Aussage gemacht: Er zeigte die Evolution des Begriffsdeutung von NoSQL auf – von „No SQL" in Abgrenzung zu etablierten Systemen, über „Not only SQL" im Sinne einer Koexistenz und der damit verbundenen Anerkennung bewährter Konzepte bis hin zu seiner Vorhersage, dass NoSQL zu „not yet SQL" werden könnte.

Diese Aussage passt gut zu den bereits beobachteten Multi-Model-Ansätzen. Man kann sehen, dass die beiden ursprünglich stark auf Abgrenzung bedachten Welten immer mehr miteinander verschmelzen – relationale Datenbanken, die dokumentorientiert arbeiten können, dokumentenorientierte Datenbanken, die für Zeitreihen genutzt werden können, NoSQL-Datenbanken die mit der Einführung von Transaktionen werben usw.

Rohdaten statt Datenbank

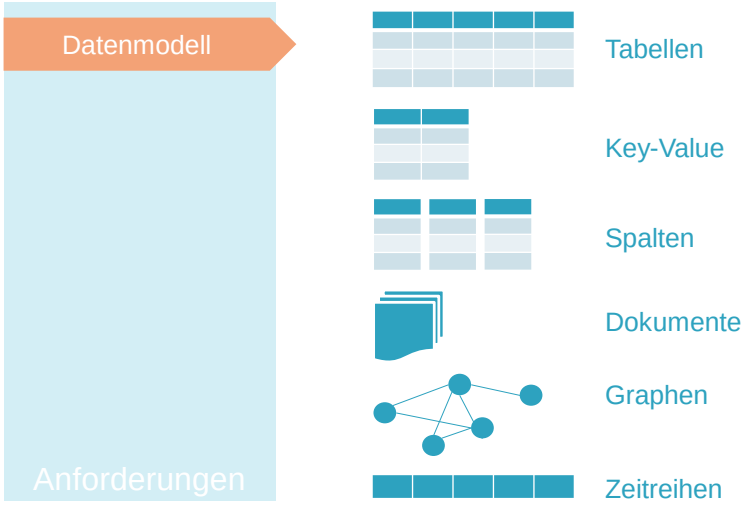


Eine Entscheidung bei der Auswahl von Datenbanken kann auch darin bestehen, große Mengen von Rohdaten (Logs, Texte, Nachrichten usw.) zu speichern, statt diese vorab zu strukturieren und dann als strukturierte Daten abzulegen. Solche sog. Data Lakes unterscheiden sich damit deutlich von sog. Data Warehouses, in denen große Mengen strukturierter Daten aus Quellsystemen zusammengeführt werden, um sie einfach auswerten zu können. Data Lakes folgen damit dem Prinzip „Schema-on-read“, d.h. erst für die Auswertung werden die Daten aus den unterschiedlichen Quellen gesammelt und in eine einheitliche Form überführt, während bei einem Data Warehouse diese Struktur vorgegeben ist („Schema-on-write“), wenn die Daten in das Data Warehouse geladen werden.

Entscheidungskriterien

Im letzten Kapitel hatten wir die Entscheidung zwischen relationalen und NoSQL-Datenbanken sehr pauschal betrachtet – Konsistenz, komplexe Abfragen und hoher Reifegrad als Argumente für die relationalen Datenbanken, horizontale Skalierbarkeit und passendere Datenmodelle für die NoSQL-Datenbanken. Die Entscheidung sollte aber sehr viel mehr Aspekte beachten.

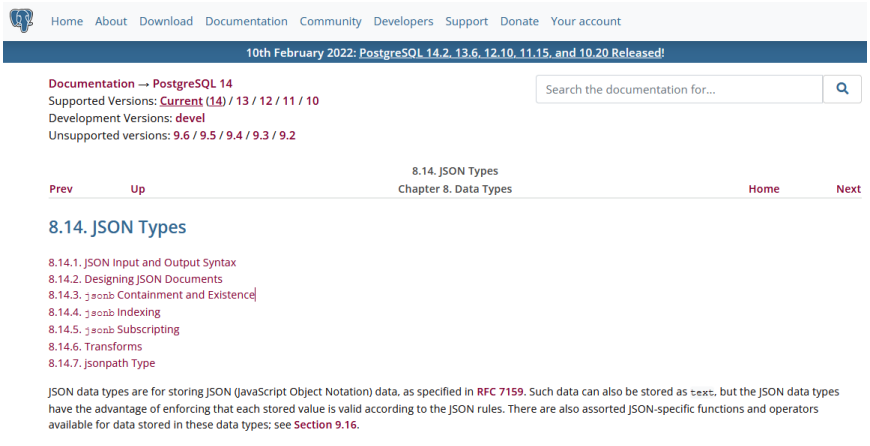
Entscheidungskriterien



Das geeignete Datenmodell kann Speicherung und Nutzung der Daten deutlich vereinfachen. Relationale Datenbanken sind für tabellarische Daten, also stark strukturierte Daten gut geeignet. Key-Value-Datenbanken bietet eine effiziente Zuordnung von Schlüssel und Werten; sie werden oft als In-Memory-Datenbanken genutzt, um Caches oder dergleichen bereitzustellen. Spaltenorientierte Datenbanken speichern die Werte spaltenweise und erlauben dadurch die effiziente Aggregation über große Datenbestände. Dokumentenorientierte Datenbanken speichern schemafrei Dokumente unter einem Schlüssel und bieten gegenüber Key-Value-Datenbanken auch Möglichkeiten, mit diesen Dokumenten zu arbeiten (Abfragen usw.). Graphen-Datenbanken sind darauf ausgelegt, Kanten und Knoten zu speichern und vernetzte Daten abzufragen. Zeitreihen-Datenbanken sind geeignet, wenn einfache Daten mit Zeitbezug effizient gespeichert und ausgewertet werden müssen.

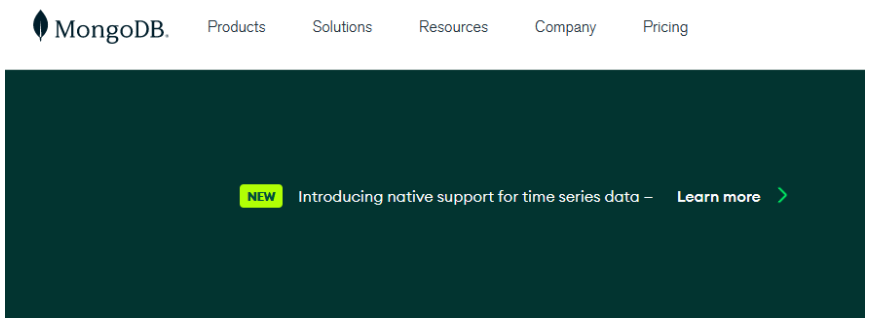
Welche Systeme sich jeweils anbieten, wurde im letzten Kapitel ausführlich dargestellt.

Vermischen der Welten

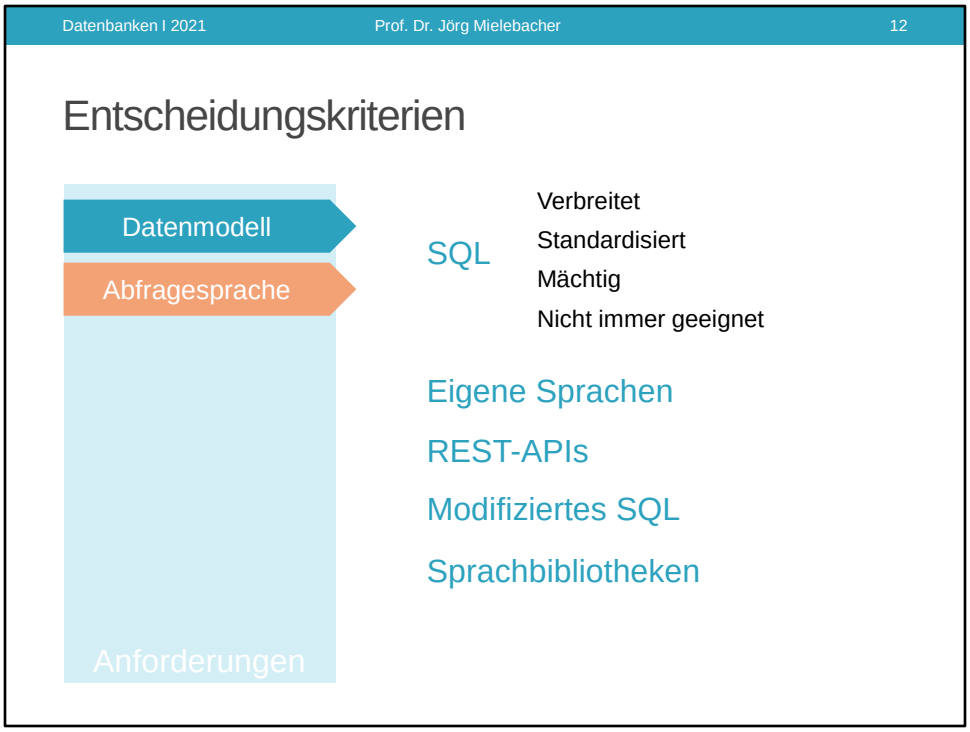


Auffällig ist die Vermischung dieser Welten, was bereits unter Multi-Model-Datenbanken dargestellt wurde. Beispielsweise unterstützen die etablierten relationalen DBMS (hier: PostgreSQL) den Umgang mit JSON- oder XML-Dokumenten. Sie bieten teilweise eigene Datentypen an und stellen Funktionen für die Verarbeitung dieser Daten bereit. Ähnlich verhält es sich auch mit anderen Datenmodellen, die in etablierten DBMS unterstützt werden (Key-Value, Zeitreihen usw.).

Vermischen der Welten



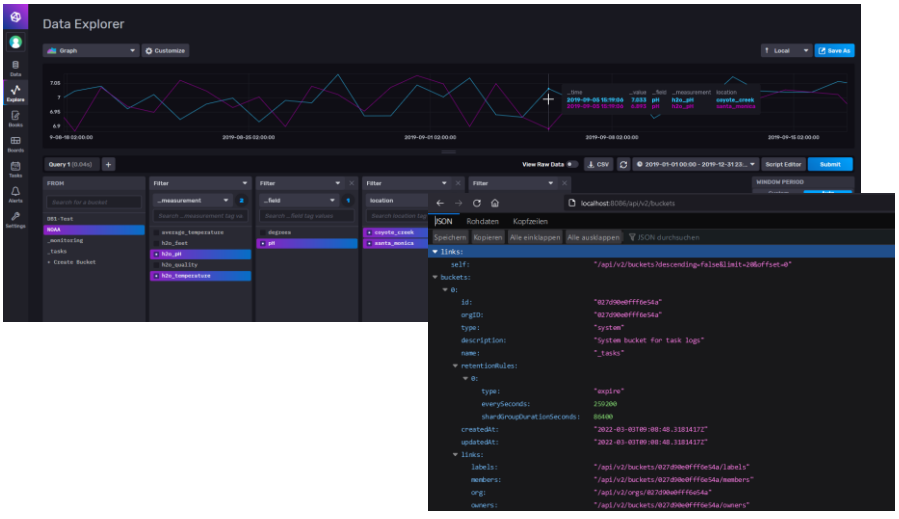
Ein weiteres Beispiel ist die Unterstützung von Zeitreihen in der dokumentenorientierten DB MongoDB. Das ist insofern besonders überraschend, als hier die Schemafreiheit als wichtige Eigenschaft zugrunde liegt; für die effiziente Handhabung von Zeitreihen ist aber eine einheitliche(re) Struktur der Daten notwendig.



Die Wahl der Abfragesprache ist ebenfalls wichtig für die Nutzung des DBMS. Relationale DBMS bieten (wie auch manch andere System) SQL, das sehr verbreitet, standardisiert und sehr mächtig ist. Allerdings stößt SQL teilweise an Grenzen, z.B. bei rekursiven Abfragen.

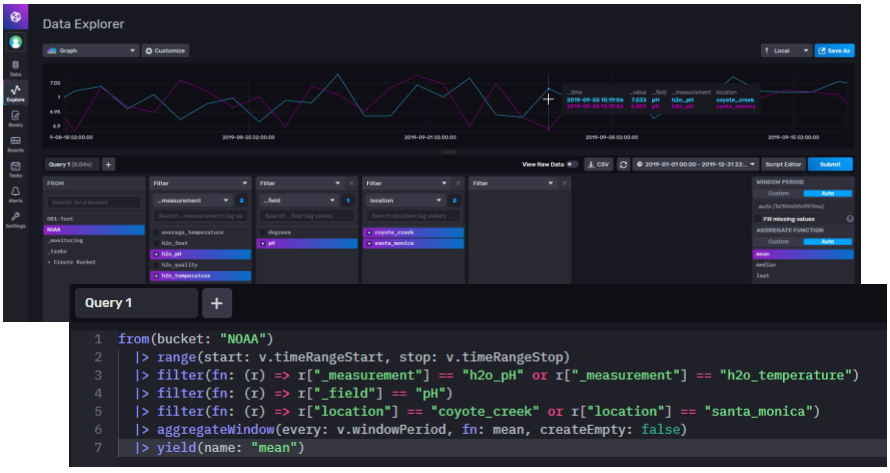
Viele DBMS stellen eigene Abfragesprachen zur Verfügung, die oftmals auf die besonderen Datenmodelle ausgerichtet sind, z.B. Cypher für die Graph-DB Neo4j. REST-APIs trifft man bei Client-Server-Datenbanken an, was es leichter macht, sie in serviceorientierte Architekturen zu integrieren, ein Beispiel ist die API der Zeitreihen-DB InfluxDB. Ganz im Sinne von „Not only SQL“ findet man oft auch Erweiterungen von SQL, ohnehin werden eigene Sprachen oft mit SQL verglichen, was angesichts der hohen Verbreitung von SQL nicht überrascht. Darüber hinaus stellen viele DBMS eigene Sprachbibliotheken für die verbreiteten Programmiersprachen zur Verfügung – hierdurch entfällt das Übersetzen der Abfragesprache in Anweisungen an das DBMS.

Beispiel: InfluxDB



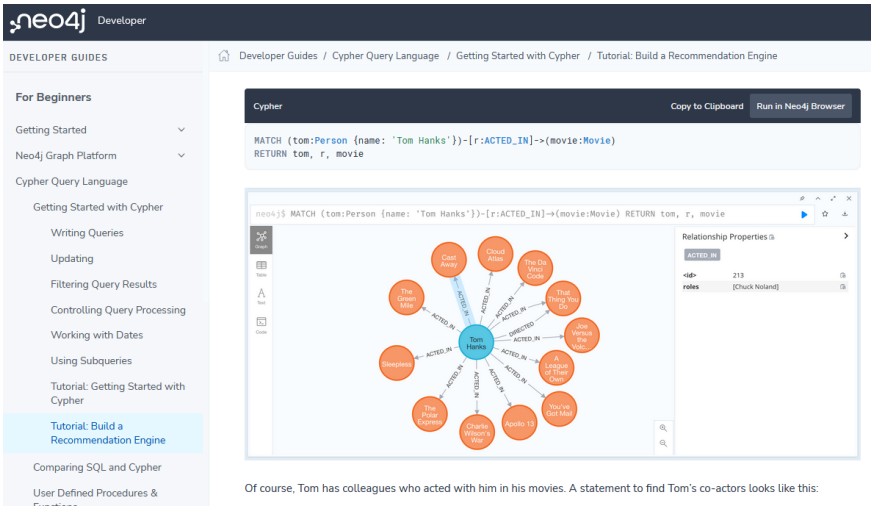
Die Zeitreihen-DB InfluxDB bietet einerseits eine web-basierte Schnittstelle, über die Abfragen auch grafisch durchgeführt werden können. Daneben bietet InfluxDB eine umfangreiche REST-API, über die Daten eingefügt und gelesen werden können.

Beispiel: InfluxDB

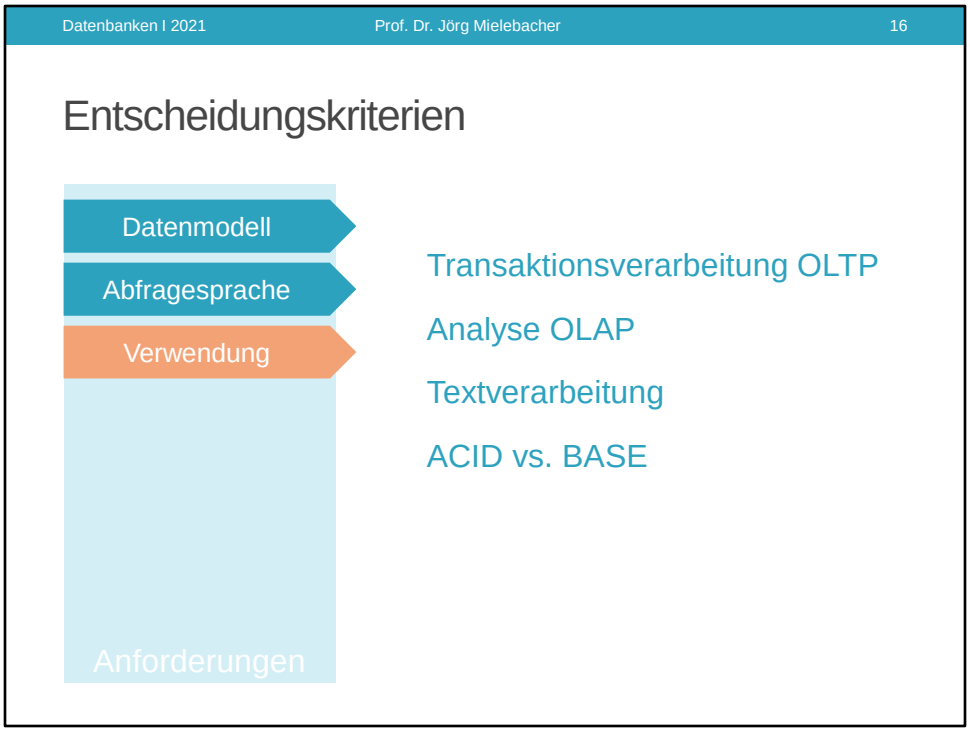


Daneben bietet InfluxDB mit FLUX eine eigene Abfragesprache für Zeitreihen an, um Zeiträume einzugrenzen, Filter anzuwenden, Aggregate zu berechnen usw.

Beispiel: Neo4j

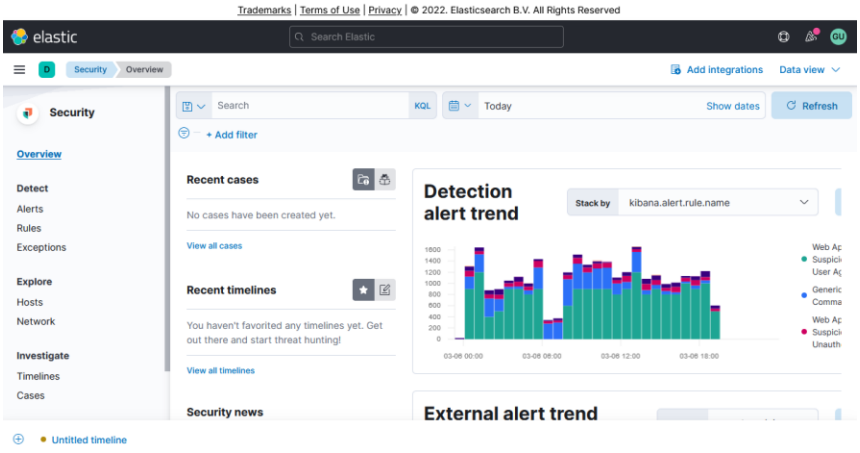


Die Graphen-DB Neo4j bietet mit Cypher eine eigene Abfragesprache an, mit der man Graphen durchsuchen kann. Die MATCH-Syntax nutzt dabei gezielt, dass es in einem Graphen Kanten zwischen Knoten gibt.

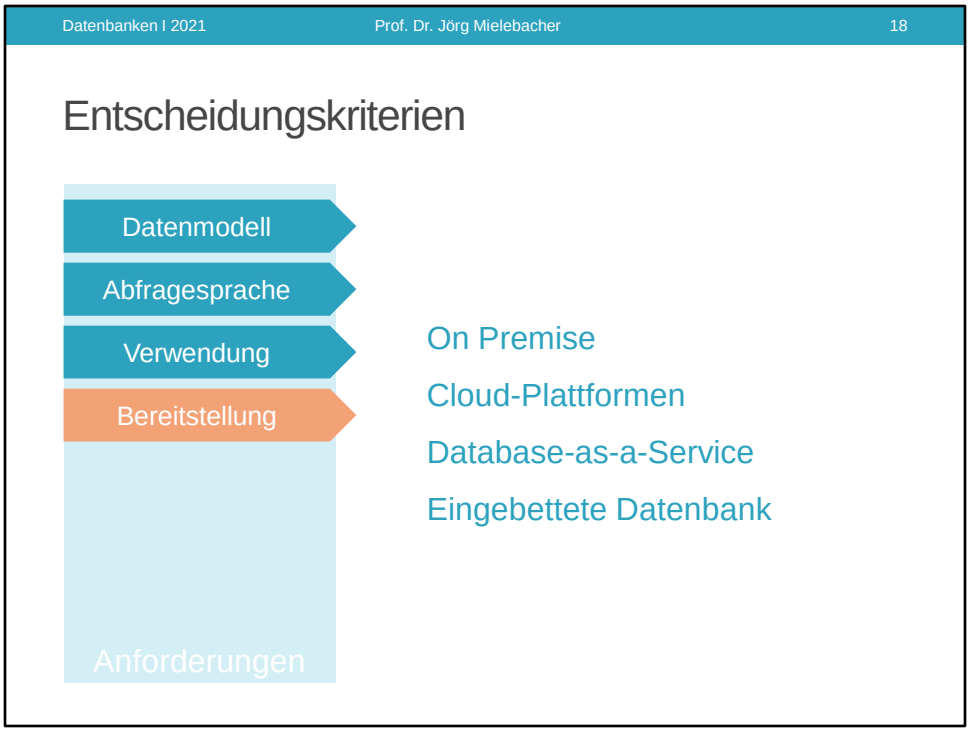


Wesentlich für die Auswahl ist auch die geplante Verwendung des DBMS: Die Transaktionsverarbeitung (Online Transaction Processing, OLTP) ist auch die schnelle Verarbeitung zahlreicher, kleiner Schreib- und Leseoperationen ausgelegt, das „Tagesgeschäft“, also z.B. Lagerbewegungen, Auftragsverwaltung, Kontobewegungen usw. Analyse (Online Analytical Processing, OLAP) wiederum zielt darauf, Datenbestände zu analysieren, um Entscheidungen zu unterstützen. Hier werden meist historische Daten gesammelt, um dann vorwiegend lesend mit komplexen Abfragen darauf zuzugreifen. Eine Besonderheit ist die Textverarbeitung, d.h. das Verwalten von unstrukturierten Daten, das Durchsuchen von Volltexten usw. Ebenfalls wesentlich sind die Anforderungen hinsichtlich ACID oder BASE, d.h. wie wichtig die Konsistenz gegenüber der Skalierbarkeit und Verfügbarkeit ist.

Beispiel: ElasticSearch



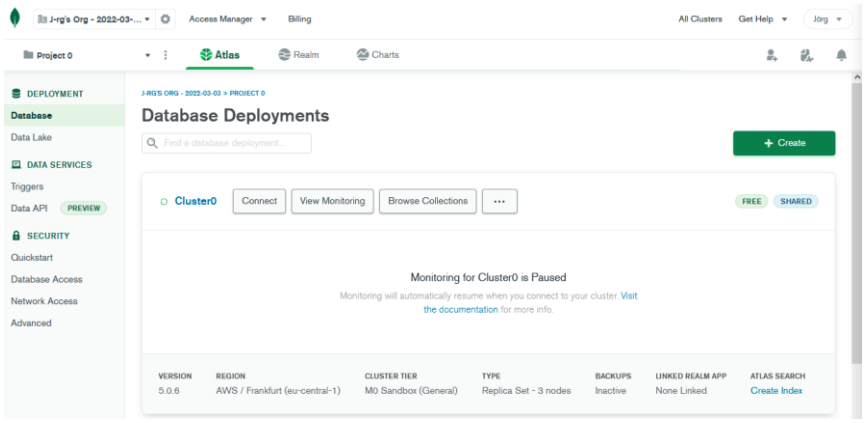
ElasticSearch ist ein beliebtes Beispiel, wenn es um Suche und Analyse geht. ElasticSearch bringt zahlreiche Werkzeuge für das Erfassung von Daten (z.B. aus Log-Dateien), das Auswerten (Berechnungen, Regeln usw.), das Durchsuchen und das Visiualisieren mit sich. Das hier dargestellte Beispiel bezieht sich auf die Auswertung von Sicherheitsvorfällen und nutzt eine typische, dashboard-basierte Web-Oberfläche.



Ein klassischer Weg der Bereitstellung ist die Installation des DBMS auf eigenen Servern, sog. On-Premise-Betrieb. Die Lagen sind dabei auf eigenen Servern gespeichert und sind von Dritten i.d.R. nicht zugreifbar. Allerdings erschwert dieser Ansatz die schnelle Skalierung, z.B. bei Kampagnen oder bei anderen, besonderen Ereignissen. So kommt es, das Serverkapazitäten teilweise unnötig (und damit teuer) vorgehalten werden, um Kapazitätsreserven für solche Fälle zu haben. Als Alternative stehen heute Cloud-Plattformen zur Verfügung (Azure, AWS usw.), die Rechenleistung und Services nach Bedarf bereitstellen. Die Datenbank liegt in diesem Fall auf Servern des jeweiligen Anbieters; deren Leistung lässt sich jedoch meist leicht über eine UI erhöhen oder verringern, was dann zu entsprechend höheren oder niedrigeren Kosten führt. Typisch für den Datenbank-Markt sind dabei cloud-basierte Database-as-a-Service-Angebote (DBaaS), durch die sich cloud-basierte DBMS nach Bedarf mieten lassen.

Während die bisherigen Ausführungen stillschweigend von Client-Server-DBMS ausgingen, darf man eine besondere Art von DBMS nicht vergessen: diejenigen, die in die Anwendung eingebettet werden, z.B. SQLite. Diese werden natürlich mit der Anwendung verteilt und erfordern damit nicht die separate Installation eines DBMS.

Beispiel: MongoDB Atlas



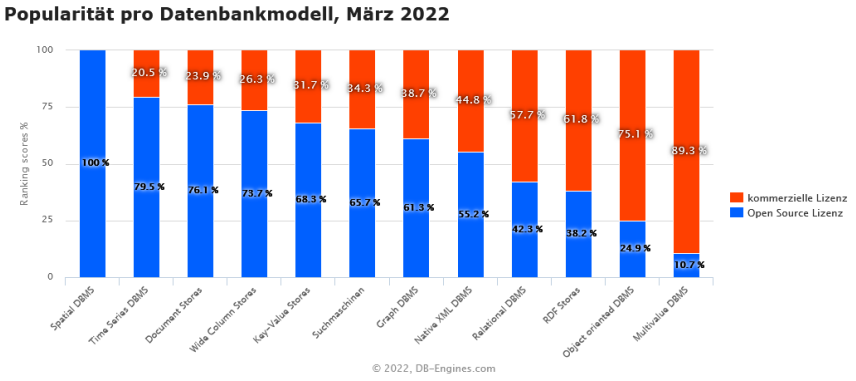
Ein Beispiel (unter vielen) ist das Cloud-Angebot von MongoDB: Mit MongoDB Atlas hat man die Möglichkeit, Datenbanken in der Cloud bereitzustellen. Auch Data Lakes und Volltextsuchen sind verfügbar. Vorteil eines solchen Modells ist die einfache Skalierung nach Bedarf, sodass man keine eigene DB-Infrastruktur vorhalten und intern betreiben muss.

Die Frage der Kosten ist häufig eng verbunden mit der Frage der Lizenzierung; für kommerzielle Produkte (z.B. Microsoft SQL Server) fallen i.d.R. hohe Lizenzgebühren an, allerdings umfassen diese meist auch entsprechende Support-Leistungen. Open-Source-Systeme stehen unter klassischen freien Lizenzen (z.B. GPL) zur Verfügung, werden mittlerweile aber oft auch unter dualen Lizenzen vertrieben. Eine gründliche Prüfung ist hier sinnvoll, insbesondere wenn das DBMS als Teil anderer Produkte (z.B. Embedded Systems) kommerziell vertrieben wird. Je nach Lizenz kann es hier nämlich notwendig werden, auch den Quellcode dieses Produkts unter eine Open-Source-Lizenz zu stellen.

Bei der Berechnung der Lizenzgebühren gibt es einmalige oder monatliche Zahlungen; sie unterscheiden sich nach Leistungsumfängen, Add-Ons, Nutzerzahlen oder auch nach Servern, Datenvolumen oder Prozessorkernen. Hinzu kommen meist Gebühren für Support-Leistungen. Bei Cloud-Angeboten basiert die Abrechnung oft auf Ist-Größen von Datenvolumen oder genutzter Rechenleistung.

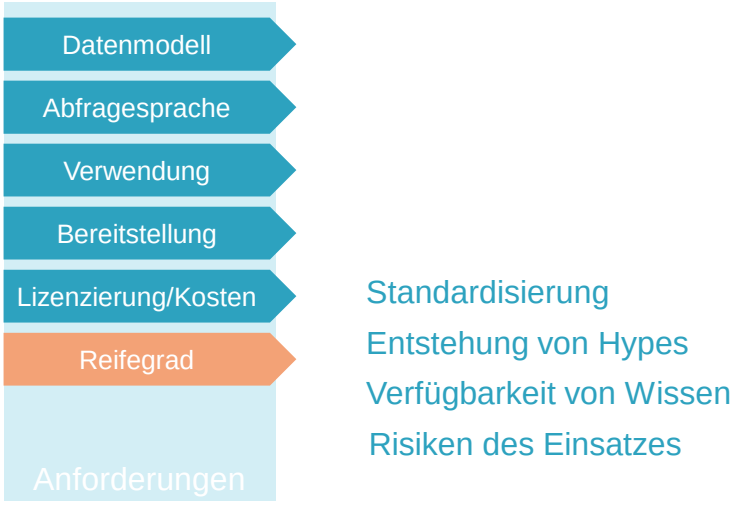
Bei On-Premise-Nutzung sollte man hinsichtlich der Kosten die Betriebskosten rund um Server, Rechenzentrum usw. nicht außer Acht lassen, insbesondere wenn hohe Verfügbarkeiten angestrebt werden, müssen Systeme redundant ausgelegt werden und muss auch die Administration entsprechend verfügbar sein.

Kommerziell vs. Open Source



Bei den DBMS ist der aktuell Markt zu etwa gleichen Teilen zwischen kommerziellen und Open-Source-DBMS aufgeteilt. Man sieht jedoch, dass je nach Typ teils erhebliche Unterschiede bestehen – während etwa bei Zeitreihen-DBMS das Verhältnis etwa 20:80 beträgt, kehrt es sich bei objektorientierten DBMS zu 75:25 um.

Entscheidungskriterien

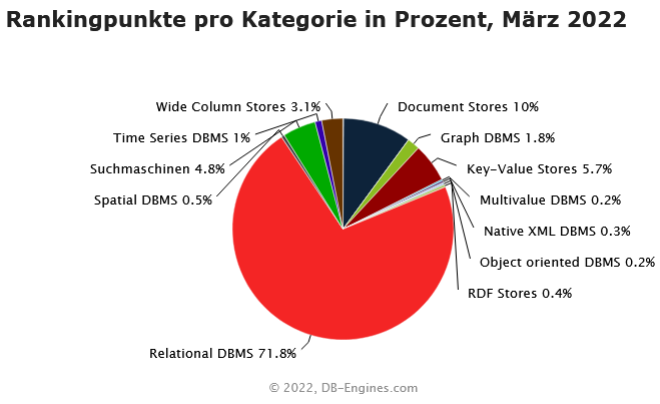


Einige der verfügbaren DBMS sind in seit vielen Jahren erfolgreich am Markt vertreten (MySQL, PostgreSQL usw.). Die starke Verbreitung und der hohe Reifegrad bringen meist mit sich, dass sehr viel Wissen in der Community verfügbar ist. Auch setzen diese etablierten Systeme meist auf Standards (z.B. SQL). Releases bringen i.d.R. keine gravierenden Veränderungen für bestehende Anwendungen mit sich. Von Nachteil kann aber sein, dass die technologische Basis dieser Systeme oft eher älter ist. Junge Systeme sind technologisch meist moderner gestaltet (z.B. verteilte Architekturen), benötigen zu Beginn aber viele Releasezyklen mit teilweise gravierenden Veränderungen. Auch unterliegen sie teilweise Hypes, denen enttäuschte Erwartungen und nachlassendes Interesse folgen; insofern ist bei jüngeren Systeme das Risiko meist höher, auf eine Technologie zu setzen, die nicht für längere Zeit in dieser Form eingesetzt werden kann. Auch gab es Fälle von Anbietern, die zunächst mit Open-Source-Lizenzen gestartet sind und dann neue Features nur für kommerzielle Lizenzen bereitstellen (z.B. ElasticSearch).

Die Community eines DBMS ist in mehrerlei Hinsicht wichtig: Einerseits bedeutet eine große Community, dass viel Wissen zu dem DBMS vorhanden ist, was bei der Suche nach geeigneten Spezialisten oder Teammitgliedern hilfreich ist, außerdem ist es die Voraussetzung, dass bei Stackoverflow usw. viele Fragen und Antworten erfasst sind, was bei der Klärung von Problemen hilft. Der Community-Support kann in solchen Plattformen erfolgen oder auch in eigenen Anbieter-Foren.

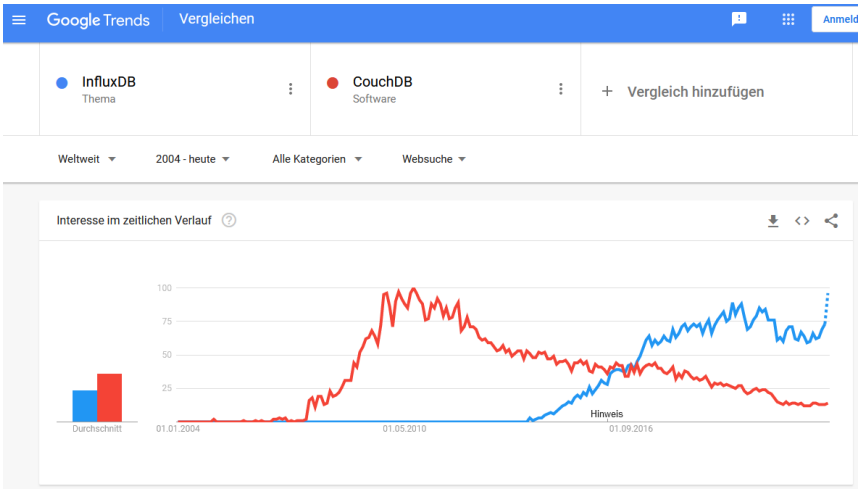
Eine große Community kann auch wichtig im Hinblick auf die Weiterentwicklung von Open-Source-Lösungen sein, wenngleich ein großer Nutzerkreis nicht zugleich einen großen Entwicklerkreis bedeutet. Hier kann es vielmehr wichtig sein, die Code-Repositories zu untersuchen, um die Größe des aktiven Entwicklerteams und den zeitlichen Verlauf der Commits zu bewerten. Ein DBMS, das nicht aktiv weiterentwickelt wird, kann sonst schnell zu einem Sicherheitsproblem werden.

Beispiel: DB-Engines Ranking



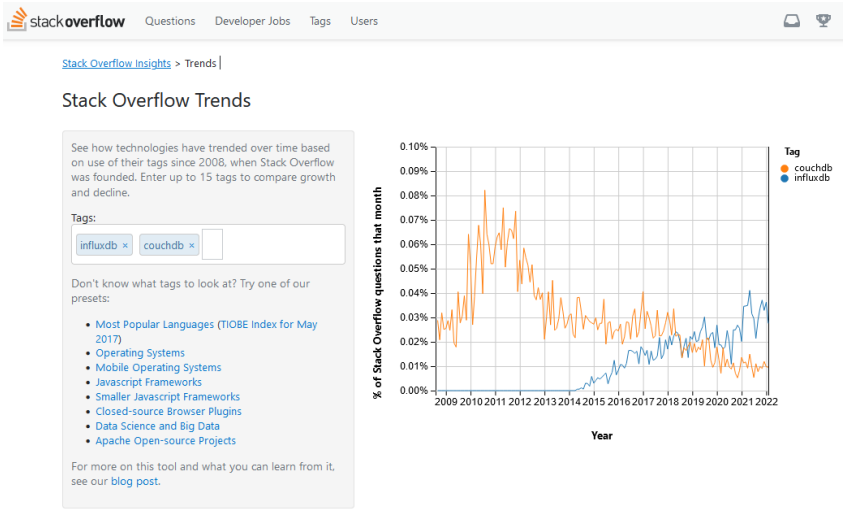
Bei der Auswahl von DBMS liefert das Ranking von DB-Engines zahlreiche Informationen, da das Ranking die Verbreitung anhand verschiedenster Indikatoren misst. Die Popularität lässt sich im Zeitverlauf sowie nach Typ, Lizenzierung usw. untersuchen.

Beispiel: Google Trends



Google Trends erlaubt es, die Häufigkeit von Suchanfragen im Zeitverlauf zu untersuchen. Da Google bei vielen Problemen während der Entwicklung ein erster Anlaufpunkt ist, liefern diese Häufigkeiten auch Hinweise auf die Verbreitung einer Technologie . Die Ergebnisse lassen sich zeitlich und regional eingrenzen, wobei die regionale Eingrenzung für die Technologieauswahl meist eher nachrangig ist. Sinkende Häufigkeiten können Ausdruck früherer Hypes sein. In jedem Fall sollte man andere Quellen hinzuziehen (z.B. Stackoverflow, Aktivitität in den Git-Repositories usw.), um zu prüfen, ob die Technologie auch langfristig eingesetzt werden.

Beispiel: Stackoverflow Insights



Stackoverflow ist eine der beliebtesten Seiten für Software-Entwickler; die Fragen spiegeln damit auch die Verbreitung und das Interesse an bestimmten Technologien wider. insights.stackoverflow.com erlaubt es, Trends zu untersuchen. So kann ein nachlassendes Interesse – ähnlich wie bei Google-Trends – ein Hinweis darauf sein, dass die Popularität einer Technologie nachlässt, was auf einen zurückliegenden Hype hinweisen kann, aber in jedem Fall auf Risiken bei der längerfristigen Nutzung deutet.

Zusammenfassung

- Die Auswahl eines DBMS muss auf der Grundlage der Anforderungen erfolgen.
- Der Entwurf muss klären, in welchem Umfang Anwendungslogik in der Datenbank abgebildet wird.
- Schemafreie Datenbanken verlagern die Entwicklung eines Schemas meist in die Anwendungsschicht.
- Data Lakes sind große Bestände von Rohdaten, die nach Bedarf ausgewertet werden.
- Die Auswahl eines DBMS hängt von zahlreichen Aspekten ab (Datenmodell, Leistung, Lizenzierung usw.).
- Technologien durchlaufen Hypes und Reifeprozesse, die man stets kritisch hinterfragen sollte.

Aufgaben

1. Was versteht man unter einem Data Lake?
2. Worauf sollte man bei der Lizenz einer Open-Source-Datenbank achten?
3. Wie lässt sich abschätzen, ob ein bestimmtes Produkt sich auch in Zukunft gut einsetzen lässt?
4. Welche Arten von Abfragesprachen trifft man in DBMS meist an?
5. Grenzen Sie OLTP und OLAP voneinander ab.
6. Welche Vor- und Nachteile bringt die Schemafreiheit einer dokumentenorientierten Datenbank mit sich?
7. Wie lassen sich JSON- oder XML-Dokumente in PostgreSQL nutzen?
8. Welche Anforderungen sind wesentlich bei der Auswahl eines DBMS?
9. Wie ist Stonebrakers „Not yet SQL“-Aussage zu deuten?
10. Finden Sie Beispiele für Anwendungslogik, die in die Datenbank verschoben wurde. Welche Vor- und Nachteile hat dies?