# Von relationalen Datenbanken zu NoSQL

Die von Don Chamberlin und Ray Boyce bei IBM erfundene Structured Query Language (SQL) ist die Standardprogrammiersprache für die Interaktion mit relationalen Datenbankverwaltungssystemen. Sie ermöglicht das Ausführen von Transaktionen wie beispielsweise dem Hinzufügen, Aktualisieren oder Löschen von Datenzeilen sowie dem Zusammenfügen von mehreren Tabellen über Primärschlüssel auf **relationalen Datenbanken** [1]. Eine relationale Datenbank speichert Daten dabei grundsätzlich immer in tabellarischer Form: Jede Zeile in der Tabelle ist dabei ein Datensatz mit einer eindeutigen ID, die als Schlüssel bezeichnet wird. Die Spalten der Tabelle enthalten die Attribute der Daten [10].

Ein großer Vorteil von relationalen Datenbanken ist es, dass diese die sogenannten **ACID-Eigenschaften** einhalten können [5, 6, 7, 10]. ACID steht für die englischen Begriffe atomicity, consistency, isolation und durability: Datentransaktionen sollen entweder ganz oder gar nicht ausgeführt werden und sind für andere Benutzer nach Abschluss der Transaktion sichtbar (Atomarität / atomicity). Vor und nach dem Ausführen der Transaktion sind alle Integritätsbedingungen erfüllt, die Daten sind also weiterhin nach zuvor definierten Regeln gültig (Konsistenz / consistency). Wenn mehrere Transaktionen gleichzeitig stattfinden, muss der Endzustand derselbe sein, als wenn die Transaktionen getrennt voneinander ausgeführt worden wären (Isolation / isolation). Und die Daten dürfen sich nur durch Transaktionen verändern, nicht aber durch äußere Einflüsse wie beispielsweise einem Stromausfall (Dauerhaftigkeit / durability). Die ACID-Eigenschaften gelten als Voraussetzung für die Verlässlichkeit von Systemen und sind ein Grund für den Erfolg relationaler Datenbanken.

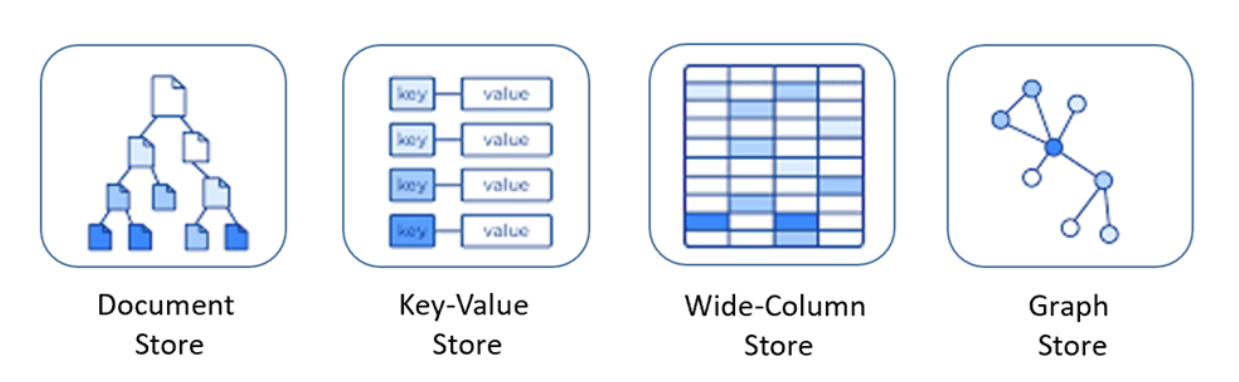
Dem entgegen steht das **Skalierungsproblem** relationaler Datenbanken: Dem Scheitern des relationalen Systems, auch bei sehr großen Datenmengen Transaktionen effizient und schnell auszuführen [3]:

Grundsätzlich gibt es drei Ansätze, um dem Skalierungsproblem entgegenzuwirken [8]: Durch Effizienzsteigerungen der Datenbank, sowie durch senkrechte und horizontale Skalierbarkeit. Beispielhafte Ansätze der Effizienzsteigerung der Datenbank sind das Speichern häufig abgefragter Daten im Zwischenspeicher (Caching) oder das Auslagern selten genutzter Daten aus der aktiven Datenbank (Archivierung). Senkrechte Skalierbarkeit bezieht sich auf die Steigerung der Leistungsfähigkeit eines einzelnen Servers, beispielsweise durch CPU-, RAM- oder Speichererweiterungen. Dem entgegen steht die horizontale Skalierbarkeit: Hierbei wird die Leistung eines Systems durch das Hinzufügen von mehreren Knoten (Servern) zu einem Cluster erweitert. Jeder Knoten im Cluster kann Anfragen verarbeiten und die Last wird auf alle Knoten im Cluster verteilt. Wenn die Anforderungen an das System steigen, können zusätzliche Knoten zum Cluster hinzugefügt werden, um die Leistung zu verbessern und die Last weiter zu verteilen.

Relationale Datenbanken sind dabei nicht auf die horizontale Skalierbarkeit als einzig endlose Skalierungsmöglichkeit ausgelegt [8]: Beim Kopieren auf Knoten muss hier die gesamte Datenstruktur übernommen werden. Dies ist notwendig, um die Konsistenz der Daten über alle Knoten hinweg sicherzustellen und sicherzustellen, dass Anfragen an den neuen Knoten dieselben Ergebnisse liefern wie Anfragen an die anderen Knoten im Cluster. Transaktionen wie das Hinzufügen einer neuen Tabellenspalte zu der relationalen Datenbank werden mit steigender Anzahl der Knoten also aufwendiger, da alle Knoten die daraus resultierende neue Struktur übernehmen müssen.

Große Unternehmen wie Facebook, Amazon und Twitter stoßen mit ihren Datenmengen an solche Skalierungsgrenzen relationaler Datenbanken und weichen deshalb auf die Alternative der NoSQL-Datenbanken, also nichtrelationalen Datenbanken aus (not only SQL) [2].

Es wird zwischen den folgenden 4 **NoSQL-Datenbanktypen** unterschieden [4, 8, 9]:

Abbildung 1: Die 4 Typen von NoSQL-Datenbanken [4]

* **Dokumentorientiert (Document Store)**: Dokumentenorientierte Datenbanken kombinieren jeden Schlüssel mit einer komplexen Datenstruktur, dem sogenannten Dokument. Jedes Dokument kann viele verschiedene Schlüssel-Wert-Paare und Schlüssel-Array-Paare und sogar eingebettete Dokumente enthalten. Dokumente zur Speicherung und Codierung von Daten werden in Standardformaten wie XML, YAML, JSON (JavaScript Object Notation) und BSON in der Datenbank gespeichert. Ein Vorteil ist, dass Dokumente innerhalb einer einzigen Datenbank unterschiedliche Datentypen haben können.
* **Schlüssel/Wert-Paar (Key-Value Store)**: Jedes einzelne Element wird in der Datenbank als Attributname (bzw. Schlüssel) zusammen mit seinem Wert gespeichert.
* **Spaltenorientiert (Wide-Column Store)**: Diese Art von Datenbank speichert Daten in Spalten statt in Zeilen. Das Vertauschen von Zeilen und Spalten hat performancetechnische Vorteile insbesondere bei Aggregationen über große Datenmengen: Da alle Werte einer Spalte zusammen gespeichert werden, müssen nur die Daten dieser Spalte gelesen werden, um Aggregationen durchzuführen, anstatt alle Zeilen lesen zu müssen.
* **Graphorientiert (Graph Store)**: Graphorientierte Datenbanken speichern Informationen über Datennetzwerke. Diese zeigen auf, wie verschiedene Datensätze zueinander in Verbindung stehen. Neo4j, RedisGraph und OrientDB sind Beispiele für Graphdatenbanken.

NoSQL-Datenbanken sind horizontal skalierbar, also durch die Hinzunahme von Servern endlos erweiterbar. Dafür verletzen NoSQL-Datenbanken fast immer mindestens eine der ACID-Eigenschaften. Dies soll anhand des folgenden Beispiels verdeutlicht werden:

Im Gegensatz zu einer relationalen Datenbank kann eine NoSQL-Datenbank problemlos auf deutsche und amerikanische Server aufgeteilt werden, sodass der deutsche Server nur deutsche Konten und der amerikanische Server nur amerikanische Konten beinhaltet [5]. Eine Transaktion könnte eine Überweisung von einem deutschen Konto auf ein amerikanisches Konto darstellen. Während dieser Verarbeitung sind die Datenbanken hierbei inkonsistent. Eine Abfrage der Datenbankinhalte während der Transaktion könnte also ergeben, dass die Überweisung auf deutscher Kontenseite schon abgeschlossen ist, bei der amerikanischen Seite aber noch nicht durchgeführt wurde. Eine solche Inkonsistenz (Verletzung des Cs in ACID) ist bei relationalen Datenbanken nicht möglich [5].

Der Unterschied zwischen relationalen und NoSQL-Datenbanken kann auch über das sogenannte **CAP-Theorem** herausgestellt werden [3, 4]: Laut dem Theorem müssen verteilte Datensysteme einen Kompromiss zwischen Konsistenz, Verfügbarkeit und Partitionstoleranz eingehen. Eine Datenbank kann dabei nur zwei dieser drei Eigenschaften gleichzeitig gewährleisten:

* **Konsistenz** bedeutet, dass jeder Knoten im Cluster die neuesten Daten zurückgibt, auch wenn dies bedeutet, dass das System die Anfrage blockieren muss, bis alle Replikate aktualisiert sind. Wenn ein konsistentes System nach einem Element abgefragt wird, das gerade aktualisiert wird, wird die Antwort verzögert, bis alle Replikate erfolgreich aktualisiert wurden. Die Antwort enthält jedoch die neuesten Daten.
* **Verfügbarkeit** heißt, dass jeder Knoten sofort eine Antwort zurückgibt, auch wenn diese Antwort möglicherweise nicht die neuesten Daten enthält. Wenn ein verfügbares System nach einem Element abgefragt wird, das gerade aktualisiert wird, gibt es die bestmögliche Antwort zum aktuellen Zeitpunkt zurück.
* **Partitionstoleranz** bedeutet, dass das System auch dann weiterhin funktioniert, wenn ein replizierter Datenknoten ausfällt oder die Verbindung zu anderen replizierten Datenknoten verliert.

Relationale Datenbanken legen in der Regel Wert auf Konsistenz und Verfügbarkeit. Sie verwenden Transaktionen, um sicherzustellen, dass die Daten konsistent bleiben, und sie sind so konzipiert, dass sie immer verfügbar sind, um Anfragen zu beantworten. Allerdings können sie anfällig für Partitionierungsprobleme sein, wenn Teile des Systems ausfallen oder die Verbindung verlieren [4].

NoSQL-Datenbanken hingegen legen in der Regel Wert auf Verfügbarkeit und Partitionstoleranz. Sie sind so konzipiert, dass sie auch dann weiterhin funktionieren, wenn Teile des Systems ausfallen oder die Verbindung verlieren, und sie sind so konzipiert, dass sie immer verfügbar sind, um Anfragen zu beantworten. Allerdings können sie in Bezug auf die Konsistenz schwächer sein als relationale Datenbanken [4].

Abbildung 2 veranschaulicht diese Beziehungen:

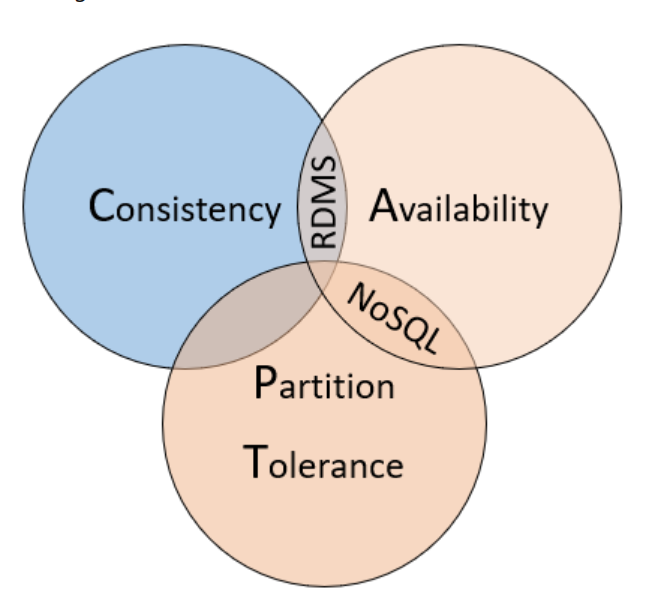


Abbildung 2: Einordnung von relationalen Datenbanken (RDMS: Relational Database Management System) und NoSQL-Datenbanken nach dem CAP-Theorem [4]

# NoSQL-Injektion

Bei Injektionsangriffen fügt ein Angreifer bösartigen Code in die Anweisungen ein, die von der Anwendung an die Datenbankschicht weitergegeben werden. Dies kann beispielsweise über Suchfelder, Formularfelder oder API-Eingaben erfolgen. Wenn die Anwendung solche Eingaben nicht bereinigt, kommt es zur Akzeptanz und Ausführung des Codes. Der Angreifer hat dann fast unbegrenzte Kontrolle über die zu Grunde liegende Datenbank: Er kann Daten verändern, einfügen oder löschen und hat Zugriff auf nichtautorisierte Daten [2]. Je nach Datenbank wird hierbei zwischen SQL- und NoSQL-Injektionsangriffen unterschieden. Die NoSQL-Injektion ist ein spezifischer Typ von Injektionsangriff, der auf NoSQL-Datenbanken ausgeführt werden kann.

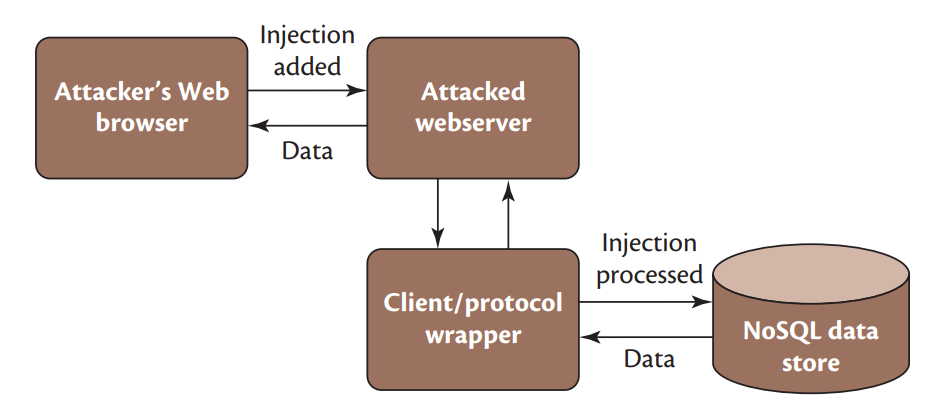


Abbildung 3: Architektur einer NoSQL-Webanwendung [2]

Zum besseren Verständnis der NoSQL-Angriffsarten zeigt Abbildung 3 die typische die typische Architektur einer Webanwendung mit NoSQL-Datenbank. Eine solche NoSQL-Datenbank könnte beispielweise mit MongoDB realisiert werden. MongoDB ist das am weitesten verbreitete NoSQL-Datenbankmanagementsystem (Stand: Oktober 2019) und wird von Großunternehmen wie eBay, Foursquare und LinkedIn angewendet. Die NoSQL-Datenbanken sind in MongoDB dokumentenorientiert. Dabei werden JSON-ähnliche Dokumente verwendet [11]. JSON steht für JavaScript Object Notation und ist ein leichtgewichtiges Datenformat, das für den Datenaustausch zwischen Anwendungen verwendet wird.

Der Zugriff auf die Datenbank erfolgt über einen Client/Protokoll-Wrapper. Ein Client/Protokoll-Wrapper ist ein Programm, das eine Schnittstelle zwischen einer Anwendung und einem Netzwerkprotokoll bereitstellt, indem es die Kommunikation zwischen den beiden vereinfacht. In Bezug auf Datenbanken kann ein Client/Protokoll-Wrapper den Zugriff auf die Datenbank vereinfachen, indem er eine Schnittstelle zwischen der Anwendung und dem Datenbankprotokoll bereitstellt [2].

In dieser Architektur kann PHP als Backend-Sprache verwendet werden, um Anfragen an die NoSQL-Datenbank zu senden und die Ergebnisse zu verarbeiten. PHP bietet eine native Unterstützung für die Arbeit mit JSON-Daten und ist daher leicht mit MongoDB kompatibel. Abbildung 4 zeigt die resultierende Architektur.

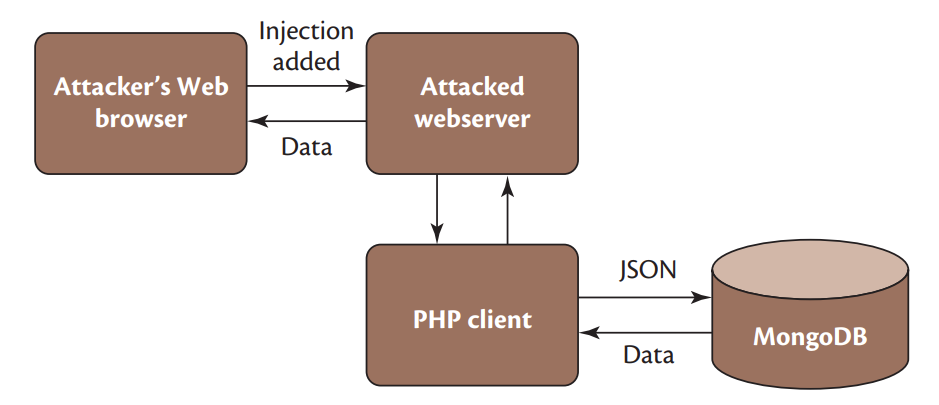


Abbildung 4: Architektur einer NoSQL-Anwendung mit MongoDB-Datenbanksystem und PHP-Client [2]

Einem Angreifer stehen je nach Art der Programmierung nun möglicherweise die folgenden NoSQL-Injektionsangriffsarten zur Verfügung:

**PHP-Tautologien mit Array-Injektionsangriffen**

Bei PHP-Tautologie-Angriffen erzeugen Angreifer durch ihre Code-Injektion Ausdrücke, die immer wahr sind (Tautologien), um Authentifizierungs- oder Zugriffsmechanismen zu umgehen. Dies wird mit dem folgenden Array-Injektionsangriff-Beispiel einer Login-Seite aufgezeigt:

Bei der Login-Seite kann ein Nutzer einen Benutzernamen und ein Passwort eingeben. PHP kodiert diese Arrays dann nativ in JSON und sendet diese vom Browser des Benutzers über HTTP POST. Ein typischen POST-URL-kodiertes Payload könnte dann folgendermaßen aussehen:

username=admin&password=1234

Diese Daten werden anschließend mit den Dokumenten aus der dahinterliegenden MongoDB-Datenbank verglichen. Es wird also abgefragt, ob ein entsprechender Nutzer mit einem korrespondierenden Passwort in der Datenbank hinterlegt ist. Dies passiert hier mittels des folgenden Befehls:

db->logins->find(array(“username”=>$\_POST[“username”], “password”=>$\_POST[“password”]));

PHP verfügt jedoch über einen eingebauten Mechanismus für assoziative Arrays, der es Angreifern ermöglicht, die folgende bösartige Injektion unter Verwendung des [$ne]-Operators zu senden:

username=admin&password[$ne]=

PHP übersetzt diese Eingabe in:

array("username" => "admin", "password" => array("$ne" => ""));

Und dies resultiert in der folgenden MongoDB-Abfrage:

db.logins.find({ username: "admin", password: { $ne: "" } })

Die Passwort-Suche wird hierbei also umgangen, indem jeder Eintrag gefunden wird, der den entsprechenden Username aufweist und ein beliebiges, nicht-leeres Passwort hinterlegt hat. Um dies zu verhindern könnten beispielsweise in der find-Methode die Eingaben vorher in Strings konvertiert werden, um Objekte wie { $ne: "" } zu unterbinden.

**Union-Abfragen**

Eine weitere Art von Angriff sind Union-Abfragen. Die Union-Abfrage ist eine bekannte SQL-Injektionstechnik, bei der Angreifer einen verwundbaren Parameter ausnutzen, um das für eine bestimmte Abfrage zurückgegebene Datenset zu ändern. Die häufigsten Verwendungen von Union-Abfragen sind das Umgehen von Authentifizierungsseiten und das Extrahieren von Daten. Hierbei kann der boolesche OR-Operatoren ausgenutzt werden, indem Ausdrücke hinzugefügt werden, die immer wahr sind (zum Beispiel eine leere Abfrage {}), was zur falschen Bewertung der gesamten Aussage führt und die illegale Datenextraktion ermöglicht.

Der folgende Entwicklercode:

string query = “{ username: ‘” + post\_ username + “’, password: ‘” + post\_passport + ‘ “ }”

könnte also mit einer solchen Injektion:

username=admin’, $or: [ {}, {‘a’: ‘a&password=’ }]

in den folgenden immer wahren Ausdruck umgewandelt werden:

{ username: ‘admin’, $or: [ {}, { ‘a’: ‘a’, password ‘’ } ]

Neben diesen beiden ausführlichen Beispielen soll hier noch auf folgende weitere NoSQL-Angriffsarten verwiesen werden:

**JavaScript-Injektionen**

Sie ermöglichen die Ausführung von JavaScript im Datenbankkontext. JavaScript ermöglicht komplizierte Transaktionen und Abfragen auf der Datenbank-Engine. Wenn nicht bereinigte Benutzereingaben an diese Abfragen weitergegeben werden, kann dies zur Injektion von beliebigem JavaScript-Code führen, was zu illegaler Datenextraktion oder -veränderung führen kann.

**Piggybacked-Abfragen**

Bei Piggybacked-Abfragen nutzen Angreifer Annahmen in der Interpretation von Escape-Sequenzen’ Sonderzeichen (wie Terminierungszeichen wie Carriage Return und Line Feed [CRLF]) aus, um zusätzliche Abfragen einzufügen, die von der Datenbank ausgeführt werden sollen. Auch dies führt zur Ausführung von beliebigem Code.

**Cross-Origin-Verletzungen**

Ein weiteres häufiges Merkmal von NoSQL-Datenbanken ist, dass sie oft eine HTTP REST-API bereitstellen, die es Client-Anwendungen ermöglicht, Datenbankabfragen durchzuführen. Datenbanken, die eine REST-API bereitstellen, sind unter anderem MongoDB, CouchDB und HBase. Die Bereitstellung einer REST-API ermöglicht eine einfache Integration der Datenbank in Anwendungen, da sie die Notwendigkeit eines Vermittlungstreibers beseitigt und es jeder Programmiersprache ermöglicht, HTTP-Abfragen an die Datenbank zu senden. Dies bietet jedoch auch ein Sicherheitsrisiko, da die REST-API die Datenbank CSRF-Angriffen aussetzt und Angreifern ermöglicht, Firewalls und andere Perimeterschutzmaßnahmen zu umgehen.

Wenn eine Datenbank in einem sicheren Netzwerk hinter Sicherheitsmaßnahmen wie Firewalls bereitgestellt wird, müssen Angreifer entweder eine Schwachstelle finden, die ihnen den Zugang zum sicheren Netzwerk ermöglicht, oder eine Injektion durchführen, die es ihnen ermöglicht, beliebige Abfragen auszuführen. Wenn eine Datenbank jedoch eine REST-API innerhalb des gesicherten Netzwerks bereitstellt, kann jeder mit Zugang zum gesicherten Netzwerk HTTP-Abfragen an die Datenbank senden. Wenn Angreifer in der Lage sind, ein HTML-Formular in eine Website einzuschleusen oder Benutzer dazu zu bringen, ihre eigene Website zu besuchen, können sie durch das Absenden des Formulars beliebige Post-Aktionen auf der Datenbank ausführen.

Um solche Angriffe zu verhindern, ist es wichtig, dass die REST-API der Datenbank sicher konfiguriert wird und dass Benutzereingaben sorgfältig validiert und bereinigt werden, bevor sie an die Datenbank gesendet werden. Es ist auch wichtig, sicherzustellen, dass nur autorisierte Benutzer Zugriff auf die REST-API haben.

[1] Chamberlin, D. D. (2012). *Early History of SQL. IEEE Annals of the History of Computing, 34(4), 78–82.* doi:10.1109/mahc.2012.61

[2] Ron, A., Shulman-Peleg, A., & Puzanov, A. (2016). *Analysis and Mitigation of NoSQL Injections. IEEE Security & Privacy, 14(2), 30–39*. doi:10.1109/msp.2016.36

[3] Schwarze Consulting GmbH. (2015). *Übersicht relational und/oder noSQL (I/II).* Data Base Camp.Abgerufen am 27.06.2023 von<https://databasecamp.de/daten/acid>

[4] Microsoft. (2022). *Vergleich der relationalen und NoSQL-Daten*. Microsoft Learn. Abgerufen am 27.06.2023 von <https://learn.microsoft.com/de-de/dotnet/architecture/cloud-native/relational-vs-nosql-data>

[5] Lang, N. (2022). *Was sind die ACID-Eigenschaften?* Data Base Camp. Abgerufen am 27.06.2023 von <https://databasecamp.de/daten/acid>

[6] Anderson, B., Nicholson, B. (2022). *SQL vs. NoSQL Databases: What's the Difference?* IBM. Abgerufen am 27.06.2023 von <https://www.ibm.com/cloud/blog/sql-vs-nosql>

[7] Anderson, B., Nicholson, B. (2022). *What is a relational database?* IBM. Abgerufen am 27.06.2023 von <https://www.ibm.com/topics/relational-databases>

[8] MongoDB. (n.d.). *NoSQL Databankenerklärung*. MongoDB. Abgerufen am 27.06.2023 von <https://www.mongodb.com/de-de/nosql-explained>

[9] Talend. (n.d.). *SQL vs. NoSQL: Wann lohnt sich welche Datenbank?* Talend. Abgerufen am 27.06.2023 von <https://www.talend.com/de/resources/sql-vs-nosql/>

[10] Oracle Deutschland. (n.d.). *Was ist eine relationale Datenbank?* Abgerufen von <https://www.oracle.com/de/database/what-is-a-relational-database/>

[11] Okman, L., Gal-Oz, N., Gonen, Y., Gudes, E., & Abramov, J. (2011). *Security Issues in NoSQL Databases. 2011IEEE 10th International Conference on Trust, Security and Privacy in Computing and Communications.* doi:10.1109/trustcom.2011.70