

# ArangoDB

## DDI Mini Challenge 2

11.11.2022

---

Etienne Roulet

GitHub Repository:

[DDI/MC2 at main · etiiiR/DDI \(github.com\)](https://github.com/etiiiR/DDI)

## Übersicht und Hauptidee

ArangoDB ist eine Multipurpose-Database mit einer Multi-Model-Implementation.

Die Bereits etablierten NoSQL-Datenbanken:

- Dokumentdatenbanken
- Key-Value Datenbanken
- Graph-Datenbanken
- Spaltenorientierte Datenbank

werden ergänzt durch die sogenannten Multimodel-Datenbanken.

- [Multimodel-Datenbanken](#)

Diese bieten mehrere Datenbanktypen in einem Technologie-Stack an. Meist baut die Architektur auf einer Key-Value-Basis auf und abstrahiert Dokumente und Graphen darauf. Die Relationen werden nicht wie in SQL mit Schüsseln relationiert, sondern mittels Graph-Relationen oder embedded List. Wie die genaue Umsetzung aussieht, ist von Datenbank zu Datenbank unterschiedlich.

Diese Arbeit wurde mit der ArangoDB implementiert. Implementierungen bei anderen Multimodel-Datenbanken wie Redis Stack, OrientDB, Apache Casandra, RethinkDB und Couchbase können sehr variieren. Auch die Funktionalitäten können von Multimodel-DB zu Multimodell-Datenbank sehr variieren.

Diese Arbeit beschäftigt sich mit dem Usecase der Multimodel Datenbanken mit Graphrelationen und schemafreien Modellen.

## Use-Cases

In der Firma Mangooa soll ein neues Bücherkaufsystem eingeführt werden. Bei den Büchern gibt es keine festen Schemen, manchmal haben Bücher mehr Informationen, manchmal weniger. Die ersten Testdaten als POC sind jedoch noch nicht so flexibel. Die Relationen zwischen den Büchern und Users sollte jeweils in einer Graph-Datenbank liegen. Mangooa verkauft Bücher an Privatpersonen und an Studenten. Mangooa hat sich das so gedacht, dass die Bücher für die Studenten per Module (courses) geordnet werden. So kann ein Schüler/Student das Modul eintragen und gleich alle passenden Bücher bestellen.

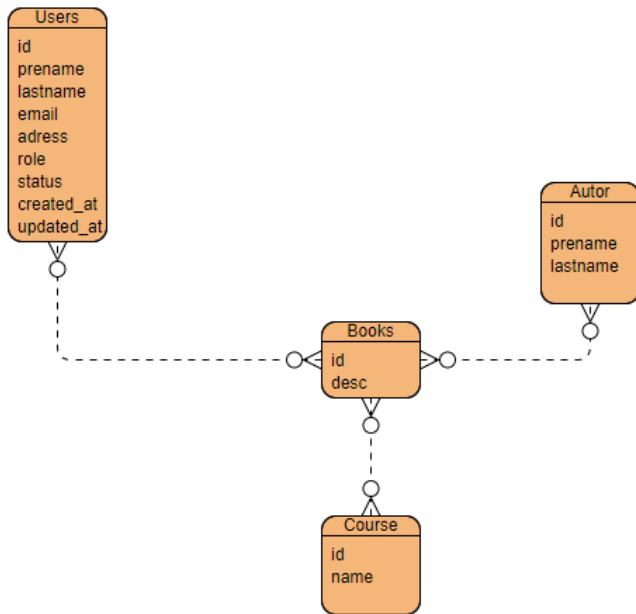
Mangooa hat sich gedacht, dass sie gerne für die Relationen eine Graph-Datenbank hätte. Jedoch haben die Daten keine komplexen Beziehungen und es wird nur ein 1..1 in und outbound Graph wirklich im Endeffekt produktiv verwendet. Deshalb haben sie euch mit dem Wunsch kontaktiert, ob ihr nicht eine Idee hättet, wie sie die Vorteile von Dokumentendatenbanken für das Speichern der Daten und für die Beziehungen Graphen verwenden könnten.

Spezifikation von Mangooa:

- Die Bücher sollten in einer Datenbank mit flexiblem Schema sein.
- Es soll eine Search Engine für Bücher und Autoren gebaut werden.
- Transaktionen sollten ermöglicht sein.
- Users können Bücher kaufen.
- Users können auch Studenten sein, diese sollen alle Bücher zu einem Modul finden und diese automatisch zusammenkaufen können.
- Die Bücher sollen in einem Ledger festgehalten werden.
- Es soll ein Graph mit allen Büchern erstellt werden, die ein User gekauft hat. Mit diesem Graph wird dann Mangooa zu einem späteren Zeitpunkt einen Recommender für Users implementieren.

Quelle: [Wie Sie eine Datenbank auswählen - Amazon Web Services \(AWS\)](#) siehe Content Management System mit Graphs Relations

## Datenmodell

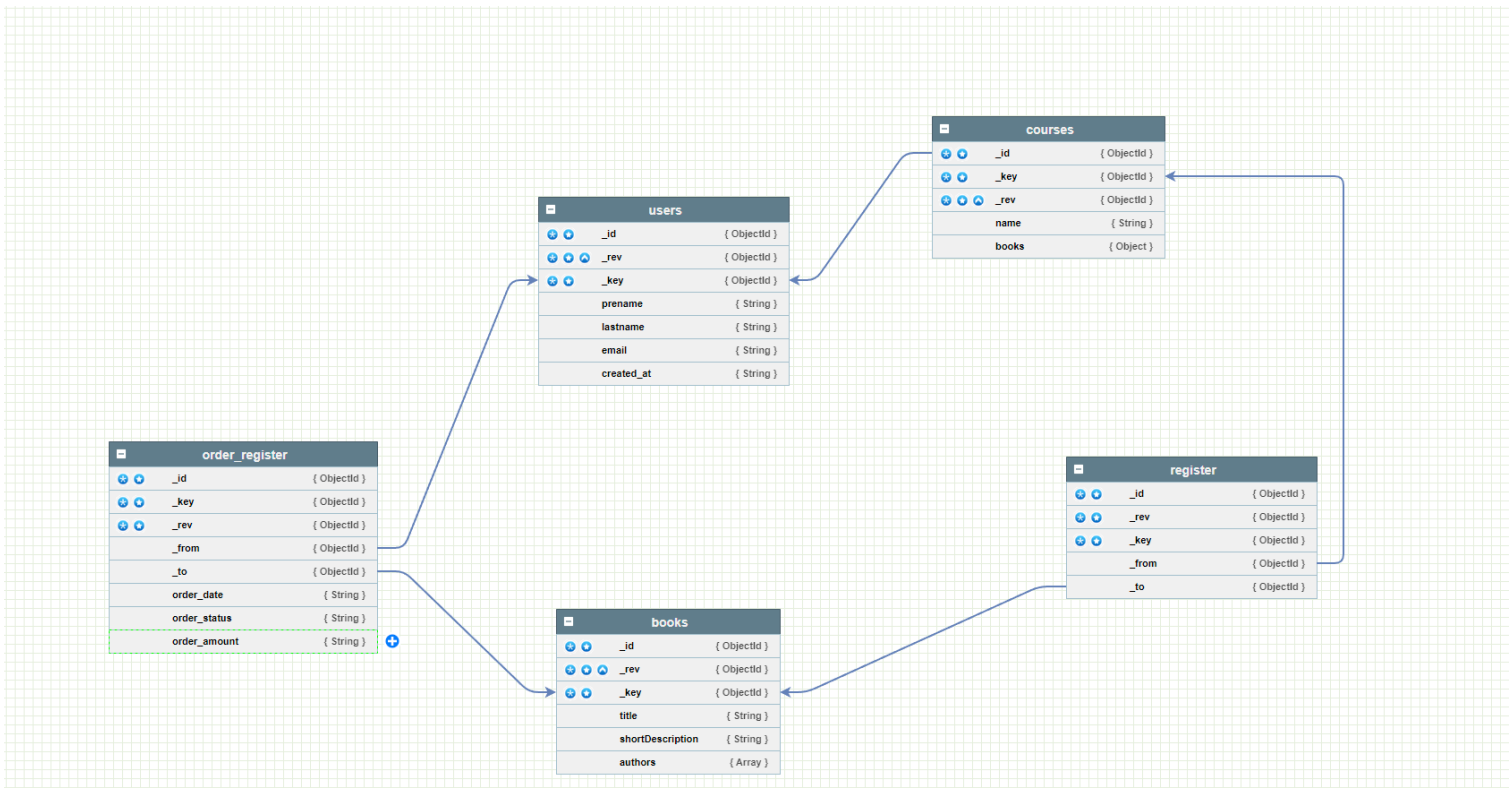


Ein User kauft ein oder mehrere Bücher, ein Buch kann auch von mehreren Usern gekauft werden.

Ein Buch wird von Autoren geschrieben, ein Autor kann mehrere Bücher schreiben.

Kurse können Bücher haben, ein Buch kann auch in mehreren Kursen vonnöten sein.

## ArangoDB Implementation:





### Erklärung:

Dass Users Bücher kaufen können, wird mit Graphrelationen zu Order abgebildet.

Dass Kurse Bücher voraussetzen, diese Relation ist jeweils im register abgelegt.

Dass Bücher Autoren haben werden, ist per embedded Array abgebildet.

Das Feld updated\_at benötigen wir nicht, da wir die Revision haben im \_rev key

([Basics and Terminology](#) | [Documents](#) | [Data Model & Concepts](#) | [Getting Started](#) | [Manual](#) | [ArangoDB Documentation](#))

## Beschreibung Implementierung

Es wurde für die Generierung der Datenbank und der Collections ein Jupiter Notebook erstellt.

Python Version: 3.10

Requirements (erster Block im Notebook script.ipynb):

```
%pip install aioarango
%pip install python-dotenv
%pip install pandas
```

Achtung im selben Ordner, in dem sich das Script.ipynb befindet, muss auch eine .env file angelegt werden:

```
arango_username=<username>
arango_password=<password>
arango_database=<database>
sys_username_arango=<username>root
sys_password_arango=<password>root
```

ArangoDB kann einfach per Docker ausgeführt werden:

Das Passwort gibt man einfach per Docker CLI mit.

Siehe [ArangoDB - Official Image | Docker Hub](#)

Ausserdem muss nun ein Datenbankuser angelegt werden und berechtigt werden.

Siehe <https://github.com/etiiiR/DDI/blob/main/MC2/readme.md>

## Dokumentation:

Installation:

Bitte Readme auf Github beachten, es müssen zwingend Environment-Variablen oder in einem lokalen .env file gesetzt werden!

<https://github.com/etiiiR/DDI/blob/main/MC2/readme.md>

Das ganze Skript ist in einem Jupiter Notebook.

Notebook:

[DDI/script.ipynb at main · etiiiR/DDI \(github.com\)](#)

## AQL Queries:

Alle Bücher anzeigen die für dieses Modul/Course benötigt wird:

for courses in 1..1 outbound "courses/GDB" register return courses

Query 3 elements 0.318 ms												JSON
_key	_id	_rev	title	isbn	pageCount	publishedDate	thumbnailUrl	shortDescription	longDescription	status	authors	categories
5063483	books/5063483	_fwkyBQm-	OSGi in Depth	193518217X	325	("\$date"/"2011-12-12T00:00:00.000-0800")	https://s3.amazonaws.com/AKIAJC5RLADLUMVRPFDQ-book-thumbnails/aves.jpg	Enterprise OSGi shows a Java developer how to develop to the OSGi Service Platform. Enterprise specification, an emerging Java-based technology for developing modular enterprise applications. Enterprise OSGi addresses several shortcomings of existing enterprise platforms, such as allowing the creation of better maintainable and extensible applications, and provide a simpler, easier-to-use, light-weight solution to enterprise software development.	A good application framework greatly simplifies a developer's task by providing reusable code modules that solve common, tedious, or complex tasks. Writing a great framework requires an extraordinary set of skills ranging from deep knowledge of a programming language and target platform to a crystal-clear view of the problem space where the applications to be developed using the framework will be used. OSGi Application Frameworks shows a Java developer how to build frameworks based on the OSGi service platform. OSGi, an emerging Java-based technology for developing modular applications, is a great tool for framework building. A framework itself, OSGi allows the developer to create a more intuitive, modular framework by isolating many of the key challenges the framework developer faces. This book begins by describing the process, principles, and tools you must master to build a custom application framework. It introduces the fundamental concepts of OSGi, and then shows you how to put OSGi to work building various types of frameworks that solve specific development problems. OSGi is particularly useful for building frameworks that can be easily extended by developers to create domain-specific applications. This book teaches the developer to break down a problem domain into its abstractions and then use OSGi to create a modular framework solution. Along the way, the developer learns software engineering practices intrinsic to framework building that result in systems with better software qualities, such as flexibility, extensibility, and maintainability. Author Alexandre Alves guides you through major concepts, such as the definition of programming models and modularization techniques, and complements them with samples that have real applicability using industry-proved technologies, such as Spring DM and Equinox.	PUBLISH	[Alexandre de Castro Alves]	[Java]
5063480	books/5063480	_fwkyBQm-	Zend Framework in Action	1933988320	432	("\$date"/"2008-12-01T00:00:00.000-0800")	https://s3.amazonaws.com/AKIAJC5RLADLUMVRPFDQ-book-thumbnails/allen.jpg	Zend Framework in Action is a comprehensive tutorial that shows how to use the Zend Framework to create web-based applications and web services. This book takes you on an over-the-shoulder tour of the components of the Zend Framework as you build a high quality, real-world web application.	From rather humble beginnings as the Personal Home Page scripting language, PHP has found its way into almost every server, corporation, and dev shop in the world. On an average day, somewhere between 500,000 and 2 million coders do something in PHP. Even when you use a well-understood language like PHP, building a modern web application requires tools that decrease development time and cost while improving code quality. Frameworks such as Ruby-on-Rails and Django have been getting a lot of attention as a result. For PHP coders, the Zend Framework offers that same promise without the need to move away from PHP. This powerful collection of components can be used in part or as a whole to speed up the development process. Zend Framework has the backing of Zend Technologies, the driving force behind the PHP programming language in which it is written. The first production release of the Zend Framework became available in July of 2007. Zend Framework in Action is a comprehensive tutorial that shows how to use the Zend Framework to create web-based applications and web services. This book takes you on an over-the-shoulder tour of the components of the Zend Framework as you build a high quality, real-world web application. This book is organized around the techniques you'll use every day as a web developer: data handling, forms, authentication, and so forth. As you follow the running	PUBLISH	[Rob Allen; Nick Le; Steven Brown]	[Web Development]

Alle Kurse anzeigen, die dieses Buch benötigen:

for books in 1..1 inbound "books/5063483" register return books

Query 2 elements 0.326 ms			
_key	_id	_rev	title
GDB	courses/GDB	_fwLXVJ2-	Grundlagen Datenbanken
DDI	courses/DDI	_fwLXVIO-	Datenbank Design

## Zentrale Stärke von Multimodel Datenbanken

### Multimodel

Eine der Hauptstärken von Multimodel-Datenbanken wie ArangoDB ist ihre Fähigkeit, verschiedene Datenmodelle in einer einzigen Datenbank zu verwalten und zu verarbeiten. Multimodel-Datenbanken sind in der Lage, verschiedene Arten von Daten wie Dokumente, graphbasierte Daten und relationale Daten in einer einzigen Anwendung zu verwalten und zu verarbeiten. Dies ermöglicht es Entwicklern, verschiedene Datenmodelle je nach den Anforderungen ihrer Anwendungen zu verwenden, ohne dass sie mehrere Datenbanken verwalten müssen.

Abschließend bieten Multimodel-Datenbanken in der Regel robuste Werkzeuge zum Replizieren und Verteilen von Daten auf mehrere Server, um die Verfügbarkeit und Leistung zu verbessern. Sie sind auch in der Regel leicht zu bedienen und bieten eine benutzerfreundliche Schnittstelle für die Arbeit mit Daten.

Quelle: [Message from Sumana P \(ArangoDB.com\)](#)

Der Nachteil ist, dass Multimodel-Datenbanken in der Regel komplexer sind als Datenbanken mit nur einem Typen.

## Zentrale Stärke von den Features der Multimodel Datenbanken

### Schemaless


Bei unserem Usecase des Büchersystems gibt es Einträge in dem books Dokument, die verschiedenen Schemen aufweisen. Hier kann ein Buch beispielsweise ein Vorgängerbuch haben oder einen Download-Link für ein E-Book. Der Vorteil von Schemaless ist, dass diese nicht geplanten Attributen in dem

Entitätstypen einfach extendiert und beschrieben werden können, ohne dass das Datenbank-Schemamodell angepasst werden muss und es somit auch auf den zyklischen oder azyklischen Release abgewartet wird, bis die neue Struktur beispielsweise dem E-Book Download link in der Tabelle persistiert werden kann.

### Graphrelationen

In Dokument basierten Datenbanken des Types Multimodel, lassen sich die Relationen der denormalisierten Dokumente in Graphrelation Dokumenten abbilden. Dies hat einen grossen Vorteil, dass innerhalb der Dokumente keine Relationen stehen und ein schlichtes und einfaches Datenmodell herrscht. Werden nun auf den Dokumenten Relationen gebraucht, wird ein zusätzliches Dokument angelegt mit den Verbindungsinformationen. Beispielsweise in ArangoDB `_from` und `_to`. So können auch beliebig viele Relationen anders abstrahiert werden. Beispielsweise ein `to_` von nur einem Ebook mit `_from` von nur





digitalen users und \_to von allen Büchern also E-Books und analoge mit \_from allen User. Dies würde nun den Zugriff erheblich erleichtern. Dabei können weitere Features immer in den Relationsdokumenten mitgespeichert werden siehe Datenbank Skript oder Screenshot auf Seite 10.

## Zentrale Stärke von ArangoDB

Die zentrale Stärke von ArangoDB ist dass es genauso ein Multimodel System implementiert, wie bereits beschrieben.

Durch die Verwendung von ArangoDB als Datenbank für das Buchverwaltungssystem könnten Entwickler verschiedene Datenmodelle wie Dokumente, graphbasierte Daten und relationale Daten verwenden, um die Daten in optimaler Struktur für die spezifischen Anforderungen des Systems zu speichern.

Darüber hinaus bietet ArangoDB umfangreiche Such- und Abfragefunktionen, die es Entwicklern erleichtern, Daten schnell zu finden und abzufragen. Dies ist für ein Buchverwaltungssystem von Vorteil, da es möglicherweise viele verschiedene Arten von Suchanfragen geben kann.

Die Declarative Query Language AQL ist der zentrale Baustein von ArangoDB. Egal ob joins, advanced path finding algorithms oder Text Search Funktionalitäten, alles wird mittels AQL abgesetzt.

Dadurch eignet sich ArangoDB perfekt für den Datenbanktyp der Multimodel Datenbanken. Was ArangoDB, jedoch nicht kann. Ist Binärdateien zu persistieren. Industriestandard ist es allemal nicht mehr Binärdateien auf den Datenbanken direkt abzulegen beispielsweise als Blob. Sondern sollte man eher einen Mediaserver in Betracht ziehen und die referenzierende URL auf der ArangoDB ablegen. Da Binärdateien meist transcodiert/konvertiert und transformiert werden müssen, was dann der Mediaserver für einen erledigt. Ausserdem will man Hypermedien Dateien meist mit einem CDN ([Content Delivery Network – Wikipedia](#)) verteilen oder mit einem IPFS ([InterPlanetary File System – Wikipedia](#)) Dienst.

## Prüfung der Stärken

### Schemaless

#### Beispiel 1 – Buch Shop Usecase:

Bücher können in ihren Schemen variieren, bei einem nicht schemafreien System müsste man das Schema auf der Datenbankmodell-Ebene anpassen und releasen. Dies ist zeitintensiv und kostspielig.

Hier mit unserer Schema freien ArangoDB documentbased Datenbank, können wir beliebig Spalten in der Datenbank befüllen.

So kann ein Eintrag wie folgt aussehen:

```
{ "name": "Lord of the Rings", "author": "J.R.R. Tolkien", "genre": "fantasy", "price": "10.00", "description":  
"The book is about" },
```

Und ein anderer:

```
{ "name": "The Hobbit", "author": "J.R.R. Tolkien", "genre": "fantasy", "price": "10.00", "description": "The  
book is about", "download_link": https://roulet.dev/12, "href_sequel": "_key" },
```

Die Zentrale Stärke liegt nun darin, dass man nicht das ganze Modell anpassen muss, nur weil die neuen E-Books neue Spalten bekommen (hier download\_link und href\_sequel).

Reflexion:

Eine schemafreie Datenbank kann viele Vorteile bringen. Meistens ist man schnell in der Umsetzung und PoC können einfach implementiert werden. Man muss sich aber gut überlegen, ob sein Datenmodell nicht besser in ein starres und schemavolles Datenmodell besser geeignet ist. Denn Schemaless bringt immer auch ein paar Nachteile mit sich wie beispielsweise:

- Integritätsprobleme, da keine festgelegte Struktur vorhanden ist, ist es schwieriger, die Integrität der Daten zu gewährleisten.
- Ausserdem ist die Datenanalyse aufwändiger, sich zuerst alle Features der Daten zu erfassen.
- Es kann dazu führen, dass Redundanzen entstehen, da es keine festen Strukturen wie es bei relationaler Datenbank der Fall ist, mit den Normalformen.
- Der Migrationsprozess zu anderen Datenbanken und Datenbanktypen kann schwieriger sein.

## Beispiel 2 – Json Struktur

Dadurch, dass kein Datenbank-Modell geplant werden muss, konnte man einfach Json-Strukturen importieren und persistieren, was dazu führt, dass man schnell eine brauchbare Datenbank ohne eine lange Planung hat.

Dokumentdatenbanken speichern meist die Daten im JSON-Format. JSON steht für JavaScript Object Notation. Also eine Textuelle Repräsentation eines JavaScript Objects. Dieses Format wird bei fast allen Programmiersprachen unterstützt und es muss lediglich, die interne Objektstruktur in JSON gedumpte, konvertiert, encoded oder serialisiert werden. Das JSON gibt somit die Struktur vor und das Schema des Eintrages. So werden die Schemen der Daten auch meist auf der Applikationsseite gelöst und validiert mittels beispielsweise Python.

Beispiel Python Serialisierung JSON:

```
import json

dictionary = {
    "id": "04",
    "name": "sunil",
    "department": "HR"
}

# Serializing json

json_object = json.dumps(dictionary, indent = 4)

print(json_object)
```

## Multimodel Graph Relationen

Relationen in Multimodel Datenbanken werden meist per Graph Relationen abgelegt. Es kann auch per embedded List in einem Document implementiert werden.

Doch laut ArangoDB ist dies nicht der State of the Art: *"If you also want to query which books are written by a given author, embedding authors in the book document is possible, but it is more efficient to use a edge collections for speed"*. (ArangoDB, 2023)<sup>1</sup>

Bei unserem Usecase konnten wir mittels Graphrelationen eine M:N-Zwischentabelle abbilden. Dies hat mehrere Vorteile. Einerseits können so Relationen schön mit einer Struktur abgebildet werden, die reproduzierbar und analysierbar ist. Bei Dokumentendatenbanken können Relationen schnell unübersichtlich werden. Denn da werden die Keys oder IDs einfach wie Fremdschlüssel mitgespeichert. Dies hat jedoch den Nachteil, dass beim Löschen in der Parent oder Child Dokumenten keine Validierung wie bei SQL geschieht. Per se weiss nämlich eine Dokumentendatabase nicht, wo sich die Relationen überall befinden. Dabei können Fehler im Schlüssel passieren, da meist keine Schlüsselvalidierungen gemacht werden, ob der

---

<sup>1</sup> <https://www.arangodb.com/docs/stable/aql/examples-join.html#using-edge-collections>

Schlüssel überhaupt in einem anderen Dokument existiert. Many to Many müssen über Array abgebildet werden.

Bei Multimodel Datenbanken kann man sich für beides entscheiden, man benötigt nicht komplexe Relationen, man kann sich gut für die embedded Relations von Dokumentbasierten Datenbanken entscheiden. In unserem Fall haben wir auch einen Array von Autoren in dem Bücher-Dokument. Da darin nur die Namen stehen und für den Buchverkauf nur die Namen wichtig sind, reicht es, eine solche einfache Relation als Array abzubilden.

Bei komplexeren Relationen kann und soll jedoch auch bei einer Multimodel Datenbank die Graphrelation verwendet werden. So können komplexe Beziehungen einfach per Nodes und Edges implementiert werden. Die Knoten stellen die Daten dar und die Kanten die Beziehungen zwischen den Knoten.

Beispiel:

Für unser Usecase des Order Ledger kann folgender Code eine Relation zwischen Buch und User darstellen:

```
await edges.insert({"_to": "users/3121802", "_from": "books/5063475", "order_date": "2020-01-01", "order_status": "open", "order_amount": 23.99})
```

Dabei konnte zusätzlich noch die Order\_date, order\_status und der order\_amount gegeben werden.

Buch und Users sind von den Relationen komplett losgelöst. Das heisst, nur der order\_register (name der Relation) weiss von den Zusammenhängen Bescheid. Was eine einfache und gute Struktur mit sich bringt und nicht so chaotisch endet wie in Dokumentdatenbanken.


The screenshot shows a database interface with a table-like view of an order register entry. The entry has the following fields:

- \_id:** order\_register/5080188
- \_rev:** \_fWlrSVu---
- \_key:** 5080188
- \_from:** books/5063475
- \_to:** users/3121802

Below the table view, there is a section titled "Select a node..." which shows a tree structure of the document. The tree structure is as follows:

- object {3}
  - order\_date : 2020-01-01
  - order\_status : open
  - order\_amount : 23.99

Ausserdem ist der grösste Vorteil natürlich, mathematische Graphematische abfragen, die bei Dokumentdatenbanken nicht so einfach umsetzbar sind. Beispielsweise finde alle User, die die Bücher für das Modul nicht gekauft haben, obwohl sie das Modul besucht haben, aber einen spezifischen Autor nicht



mögen. So könnte Mangoa ihren Recommender mittels des Graphen abfragen den Usern ein alternatives Buch anbieten, weil sie den Autor nicht mögen, jedoch das Modul besuchen, welches ein Buch dieses Autors voraussetzt.

Und natürlich, auch wenn wir es in diesem Usecase nicht verwenden, gibt es viele ziemlich nützliche mathematische Graphtheoretische Abfragen, die bei dokumentbasierten Datenbanken meist fehlen. Wie beispielsweise shortest path, welches die Abfrage mit der geringsten Anzahl von Kanten zwischen zwei Knotenpunkten macht. Was beispielsweise bei Geoapplikationen sehr von Vorteil ist, um die kürzeste Route von einem Ort zum anderen zu berechnen.

## Reflexion

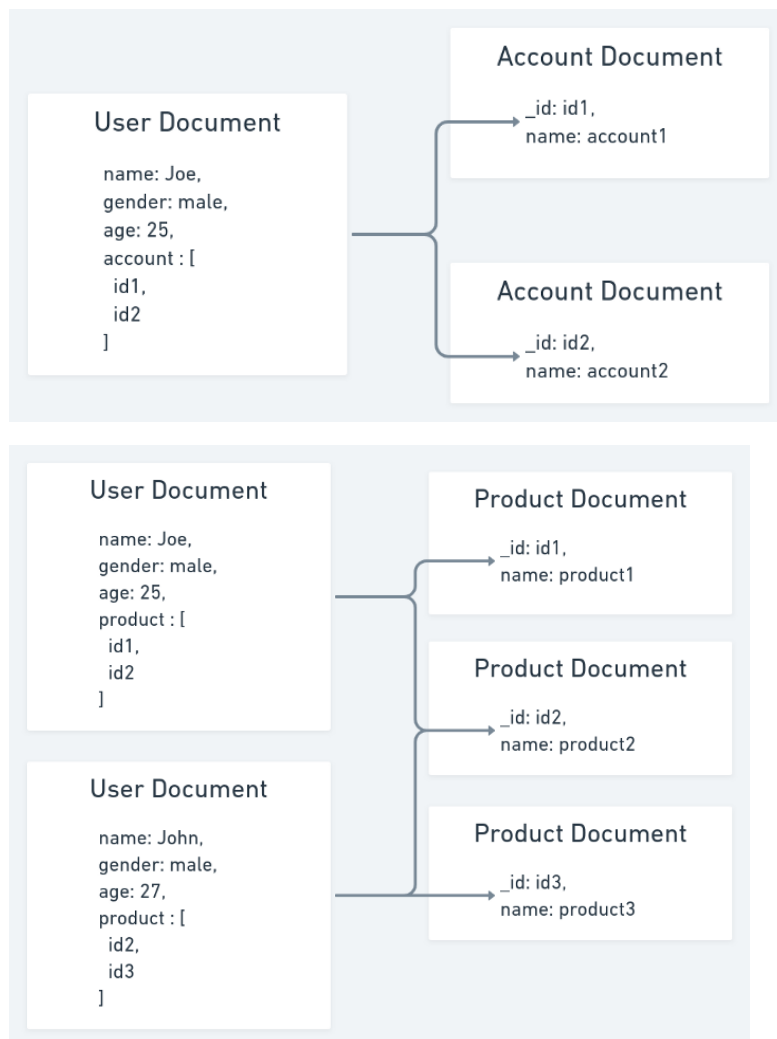
Mittels den Graph-Relationen konnten die Relationen zwischen den verschiedenen Dokumenten perfekt abgebildet werden. Ich bin überzeugt, dass Multimodel-Datenbanken oder vielleicht auch NewSQL-Datenbanken wie SurrealDB die Zukunft sein werden. Denn es vereint das Beste aus mehreren Datenbankwelten.

## Vergleiche

### Vergleich mit Dokument basierende Datenbanken

Reine dokumentbasierte Datenbanken eignen sich ebenfalls bestens. Sie haben nur nicht die Flexibilität der Multimodel-Datenbanken. Hier können die Relationen entweder embedded werden und denormalisiert dargestellt oder per Schlüssel in einem anderen Dokument hinterlegt werden. Hier gibt es natürlich den Nachteil, dass der Schlüssel als Liste hinterlegt werden muss, wenn die Beziehungen 1:m oder m:n sind.

Dennoch eignen sich dokumentbasierende Datenbanken auch für den Usecase. Wirklich schön finde ich dies jedoch nicht und prognostisch meinen Fuss ins Feuer legen, auf dass eine gute Datenqualität herrschen würde, würde ich auch nicht unbedingt.



Quelle: Bild Medium<sup>2</sup>

<sup>2</sup> <https://visrozar.medium.com/mongodb-handling-relations-3eb799403c8>

## Vergleich mit Graphdatenbanken

Multimodel-Datenbanken haben mit Graphrelationen und Graphdatenbanken einige Gemeinsamkeiten. Da Multimodel-Datenbanken meist, wie Graph-Datenbanken funktionieren.

Multimodel Datenbanken eignen sich mehr für Kundendaten und Daten, die sowohl strukturierte als auch viele unstrukturierte Daten-Elemente enthalten. Zum Beispiel könnten Stammdaten einer fest strukturierten form nachgehen und freiere Texte wie Kommentare und Notizen in unstrukturierten Formen persistiert werden. Die Relationen können dann gleich wie in einer Graph-Datenbank abgelegt werden, beispielsweise wie ein Kommentar zu dem User (Stammdaten) abgelegt sind.

In reinen Graph Datenbanken lässt sich dabei nicht so flexibel umgehen und sie werden meist für reine Beziehungsdaten, den Verbindungen zwischen den Elementen genutzt. Graphdatenbanken sind stark, um Netzwerk darzustellen, wie Freundschaften, Familienbeziehungen und berufliche Beziehungen. Eine Graph Datenbank wäre in der Lage, diese Beziehungen zu speichern und schnell Abfragen darüber zu stellen, wer mit wem verbunden ist und wie diese Verbindungen gestaltet sind.

Eine Multimodel-Datenbank macht sich genau diesen Usecase einer Graph-Datenbank zu seinem eigenen Vorteil, dass es die Relationen der Daten genauso in Graphen abstrahiert.

## Vergleich mit Timeseries Datenbanken

Unser Usecase mit dem Buchshop ist klar für Timeseries Datenbanken ungeeignet. Denn eine Zeitreihendatenbank, wie es der Name auch schon vermuten lässt, ist eine speziell für zeitgestempelte Daten entwickelt. Sie sind besonders nützlich bei Daten, die man über einen bestimmten Zeitraum hinweg analysieren möchte. Eine Timeseries-Datenbank eignet sich jedoch weniger für zeitstempellose Daten oder die nicht über einen bestimmten Zeitraum hinweg analysiert werden müssen.

Wir haben zwar bei unserem Modell auch Zeitstempel wie `created_at`. Diese Zeitstempel haben jedoch nur den Zweck, Metainformationen für die Applikation oder den Administrator der Applikation zu hinterlassen. Sie dienen nicht zur Analyse über einen Zeithorizont hinweg.

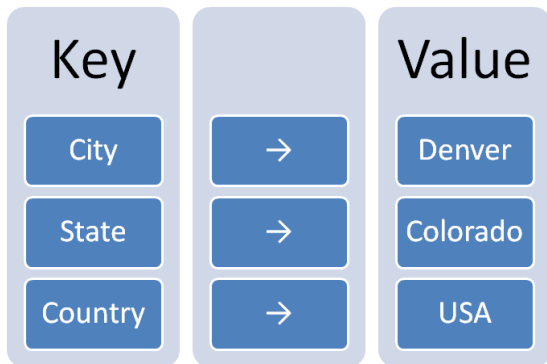
Typischerweise haben Zeitreihendatenbanken auch den Index auf den Timestamps. Dies erlaubt das effiziente Arbeiten mit Zeitperioden, um die Trendänderung über die Zeit zu verfolgen.

## Vergleich mit Key-Value Stores

Key Value Stores erlauben es den Nutzern der Datenbank, schnell Daten zu lesen und Daten zu schreiben, darum werden sie oft als Caching-Strategie bei Grosskonzernen in riesigen Applikationen verwendet. Ausserdem werden oft auch Session Daten in Key Value Stores abgelegt.

Key Value Stores sind jedoch nicht dafür ausgelegt, komplexe Schemata als schemavolles und schema-freies System zu behandeln. Denn Key Value Stores werden rein nur über den Schlüssel gequeryed.

Also zu jedem einzigartigen Key wird ein entsprechendes Value hinterlegt.



Ausserdem ist ein Key Value Store darauf ausgelegt, dass die Daten, die in dem Store abgelegt sind, in einer frequenten Nutzung verwendet werden.

Ein anderer Punkt ist auch, dass in Key Value Stores keine Relationen abgespeichert werden können wie in Dokument oder Graph Datenbanken. Dies macht für uns ein Key Value Store unbrauchbar.

## Bibliography

ArangoDB. (2023, 01 01). *ArangoDB.com*. Retrieved from  
<https://www.ArangoDB.com/docs/stable/aql/examples-join.html>:  
<https://www.ArangoDB.com/docs/stable/aql/examples-join.html>

Quelle: [Performance of ArangoDB](#)

Quelle: [Message from Sumana P \(ArangoDB.com\)](#)

[ArangoDB – A different approach to NoSQL \(slideshare.net\)](#)