B1E

Ich kann ein Datenmodell für eine NoSQL Datenbank entwerfen.

Fragenstellung und Lernziele

Fragenstellung: Wie entwirft man ein flexibles und performantes Datenmodell für eine NoSQL-Datenbank?

Lernziele:

- Verständnis der Kernkonzepte des NoSQL-Datenmodell-Designs, insbesondere im Kontext von Dokumentendatenbanken.
- Fähigkeit, die Anforderungen einer Anwendung in ein geeignetes NoSQL-Datenmodell zu übersetzen.
- Analyse und Begründung von Designentscheidungen wie Denormalisierung, Einbettung (Embedding) vs. Referenzierung (Linking).
- Erstellung eines beispielhaften Datenmodells (z.B. im JSON-Format)
- Bewertung der Auswirkungen des Datenmodells auf typische Abfragemuster und Skalierbarkeit

Umsetzung

Kernkonzepte des NoSQL-Datenmodell-Designs (Dokumentendatenbanken)

Im Gegensatz zu relationalen Datenbanken, die auf normalisierten Tabellen mit festen Schemata basieren, bieten NoSQL-Dokumentendatenbanken (wie MongoDB oder Couchbase) mehr Flexibilität. Die Kernkonzepte umfassen:

1. Aggregat-Orientierung: Daten, die zusammengehören und häufig gemeinsam abgerufen werden, sollten idealerweise in einem einzigen Dokument (Aggregat) gespeichert werden. Dies minimiert die Notwendigkeit von Joins.
2. Schema-Flexibilität: Dokumente in derselben Sammlung müssen nicht dieselbe Struktur haben. Felder können hinzugefügt oder entfernt werden, ohne das gesamte Schema zu ändern. Dies ist vorteilhaft für agile Entwicklung und sich entwickelnde Anforderungen.
3. Priorisierung von Leseleistung: NoSQL-Datenmodelle werden oft so optimiert, dass die häufigsten Lesezugriffe sehr schnell erfolgen, auch wenn dies zu einer gewissen Datenredundanz (Denormalisierung) und komplexeren Schreibvorgängen führen kann.

4. Skalierbarkeit: Durch die Verteilung von Daten auf viele Server (Sharding) können NoSQL-Datenbanken horizontal skalieren. Das Datenmodell sollte dies unterstützen.

Anforderungen einer Anwendung in ein NoSQL-Datenmodell übersetzen

Der Prozess beginnt mit dem Verständnis der Anwendungsanforderungen:

1. Entitäten identifizieren: Welche Hauptobjekte gibt es in Ihrer Anwendung (z.B. Benutzer, Produkte, Bestellungen)?
2. Beziehungen definieren: Wie stehen diese Entitäten zueinander (z.B. ein Benutzer hat viele Bestellungen, eine Bestellung enthält viele Produkte)?
3. Abfragemuster analysieren: Welche Daten werden am häufigsten zusammen gelesen? Welche Filter und Sortierungen werden benötigt? Wie sehen typische Schreibvorgänge aus?
4. Performance- und Skalierbarkeitsziele: Wie viele Lese-/Schreibvorgänge pro Sekunde werden erwartet? Wie gross werden die Datenmengen?

Beispielanwendung: Ein einfaches Blog-System mit Benutzern, Blog-Posts und Kommentaren.

Entitäten: User, Post, Comment

Beziehungen:

- Ein `User` kann viele `Posts` schreiben.
- Ein `User` kann viele `Comments` schreiben.
- Ein `Post` wird von einem `User` geschrieben.
- Ein `Post` kann viele `Comments` haben.

Typische Abfragemuster:

- Anzeige eines Posts mit allen zugehörigen Kommentaren und Autoreninformationen.
- Auflistung aller Posts eines bestimmten Benutzers.
- Anzeige der neuesten Posts.

Designentscheidungen: Denormalisierung, Einbettung vs. Referenzierung

Basierend auf den Anforderungen werden Entscheidungen über die Datenstruktur getroffen.

Denormalisierung

Hierbei werden redundante Datenkopien gespeichert, um Lesezugriffe zu beschleunigen, indem Joins vermieden werden.

- **Beispiel**: Statt nur die `user_id` im Post-Dokument zu speichern, könnte man auch den

`username` direkt im `Post`-Dokument speichern. Wenn der Benutzername angezeigt werden soll, muss nicht extra das `User`-Dokument geladen werden.

- **Nachteil**: Bei Änderungen (z.B. Benutzername ändert sich) müssen alle redundanten Kopien aktualisiert werden, was Schreibvorgänge komplexer macht.

Einbettung (Embedding) vs. Referenzierung (Linking)

Dies ist eine zentrale Entscheidung im Dokumentenmodell-Design.

Einbettung (Embedding): Verknüpfte Daten werden direkt in das übergeordnete Dokument integriert.

Beispiel: Kommentare (Comments) könnten als Array direkt in das Post-Dokument eingebettet werden.

```
{
  "_id": "post123",
  "title": "Mein erster Post",
  "content": "...",
  "author_id": "userABC",
  "author_username": "Alice", // Denormalisiert
  "comments": [
    { "commenter_id": "userXYZ", "commenter_username": "Bob", "text": "To
    { "commenter_id": "userDEF", "commenter_username": "Charlie", "text":
  ]
}
```

Vorteile	Nachteile
Daten können in einer einzigen Leseoperation abgerufen werden (Post + Kommentare)	Dokumente können sehr gross werden, was die Performance beeinträchtigen und zu Grössenbeschränkungen (z.B. 16MB in MongoDB) führen kann
Atomare Operationen auf dem Hauptdokument sind einfacher (wenn die Datenbank dies unterstützt)	Das Aktualisieren eines einzelnen eingebetteten Elements (z.B. eines Kommentars) erfordert das Neuschreiben des gesamten Hauptdokuments
	Das separate Abfragen der eingebetteten Elemente (z.B. alle Kommentare eines bestimmten Benutzers über alle Posts hinweg) ist oft ineffizient oder unmöglich

Referenzierung (Linking): Verknüpfte Daten werden in separaten Sammlungen (Collections) gespeichert, und im Hauptdokument wird nur eine Referenz (z.B. die ID) auf das verknüpfte Dokument gespeichert.

Beispiel: Kommentare werden in einer eigenen Comments-Sammlung gespeichert, und im Post-Dokument wird nur eine Liste von comment_ids oder gar keine direkte Referenz gehalten (Kommentare referenzieren den Post).

Posts Collection:

```
{
  "_id": "post123",
  "title": "Mein erster Post",
  "content": "...",
  "author_id": "userABC",
  "author_username": "Alice", // Denormalisiert
  "comment_count": 2 // Denormalisiert, optional
}
```

Comments Collection:

```
[
  {
    "_id": "comment789",
    "post_id": "post123",
    "commenter_id": "userXYZ",
    "commenter_username": "Bob",
    "text": "Toller Post!",
    "date": "..."
  },
  {
    "_id": "comment790",
    "post_id": "post123",
    "commenter_id": "userDEF",
    "commenter_username": "Charlie",
    "text": "Interessant.",
    "date": "..."
  }
]
```

Vorteile	Nachteile
Dokumente bleiben kleiner und übersichtlicher	Um alle Daten zu erhalten (z.B. Post und seine Kommentare), sind mehrere Abfragen an die Datenbank notwendig (Application-Level Join)
Verknüpfte Daten können unabhängig voneinander abgefragt und modifiziert werden	Einige Datenbanken bieten $lookup (MongoDB) für serverseitige Joins, was aber die Komplexität der Abfrage erhöht

Besser geeignet für “many-to-many”-Beziehungen oder wenn die Anzahl der verknüpften Elemente sehr gross sein kann	

Entscheidungsfindung:

- **“One-to-few” Beziehungen**: Einbettung ist oft gut, wenn die Anzahl der eingebetteten Elemente begrenzt ist und sie meist zusammen mit dem Elternelement gelesen werden (z.B. Adressen eines Benutzers).
- **“One-to-many” / “One-to-squillions” Beziehungen**: Referenzierung ist meist besser, besonders wenn die Anzahl der “many”-Elemente gross ist oder sie oft unabhängig vom Elternelement bearbeitet/abgefragt werden (z.B. Kommentare zu einem populären Blog-Post).
- **Häufigkeit des Zugriffs**: Werden die Daten fast immer zusammen benötigt? Dann Einbettung. Werden sie oft getrennt benötigt? Dann Referenzierung.
- **Datenkonsistenz**: Bei Denormalisierung oder Zählern (wie comment_count) muss die Konsistenz bei Schreibvorgängen sichergestellt werden (z.B. durch Transaktionen, falls unterstützt, oder durch eventual consistency).

Beispielhaftes Datenmodell (JSON-Format)

Für unser Blog-System entscheiden wir uns für eine Mischung: Wir referenzieren Kommentare, da ein Post potenziell sehr viele Kommentare haben kann, und denormalisieren einige Autoreninformationen für eine schnelle Anzeige.

1. Users Collection (users)

```
[
  {
    "_id": "userABC",
    "username": "Alice",
    "email": "alice@example.com",
    "join_date": "2023-01-15T10:00:00Z"
  },
  {
    "_id": "userXYZ",
    "username": "Bob",
    "email": "bob@example.com",
    "join_date": "2023-02-20T14:30:00Z"
  }
]
```

2. Posts Collection (posts)

```
[
  {
    "_id": "post123",
    "title": "Einführung in NoSQL",
    "content": "NoSQL-Datenbanken bieten viele Vorteile...",
    "author_id": "userABC", // Referenz zum User
    "author_username": "Alice", // Denormalisiert für schnelle Anzeige
    "created_at": "2024-06-01T12:00:00Z",
    "tags": ["NoSQL", "Datenbanken", "Tutorial"],
    "comment_count": 1 // Denormalisierter Zähler, optional
  },
  {
    "_id": "post456",
    "title": "Datenmodellierung für Performance",
    "content": "Ein gutes Datenmodell ist entscheidend...",
    "author_id": "userXYZ", // Referenz zum User
    "author_username": "Bob", // Denormalisiert für schnelle Anzeige
    "created_at": "2024-06-05T09:30:00Z",
    "tags": ["Datenmodellierung", "Performance"],
    "comment_count": 0
  }
]
```

3. Comments Collection (comments)

```
[
  {
    "_id": "comment789",
    "post_id": "post123", // Referenz zum Post
    "commenter_id": "userXYZ", // Referenz zum kommentierenden User
    "commenter_username": "Bob", // Denormalisiert
    "text": "Sehr informativer Artikel, Alice!",
    "commented_at": "2024-06-01T15:20:00Z"
  }
]
```

Bewertung der Auswirkungen des Datenmodells auf typische Abfragemuster und Skalierbarkeit

Abfragemuster

Anzeige eines Posts mit Autoreninfo und Kommentaren:

1. Lade den Post aus `posts` anhand der `_id`. (1 Abfrage)

- `author_username` ist bereits im Post-Dokument enthalten (denormalisiert).
- Lade alle Kommentare aus `comments` bei denen `post_id` der ID des geladenen Posts

- Lade alle Kommentare aus `comments`, bei denen `post_id` der ID des geladenen Posts entspricht. (1 Abfrage, Index auf `comments.post_id` notwendig)
- `commenter_username` ist bereits im Kommentar-Dokument enthalten.
- Performance: Zwei schnelle, indizierte Abfragen. Dies ist performanter als viele Joins in einer relationalen DB für denselben Anwendungsfall und vermeidet das Problem riesiger Dokumente bei Einbettung.

2. Auflistung aller Posts eines bestimmten Benutzers (z.B. “Alice”):

- Suche in `posts` nach `author_id`: “userABC”. (1 Abfrage, Index auf `posts.author_id` notwendig)
- Performance: Schnell und effizient.

3. Anzeige der neuesten 10 Posts:

- Suche in `posts`, sortiere nach `created_at` absteigend, limitiere auf 10. (1 Abfrage, Index auf `posts.created_at` notwendig)
- Performance: Schnell und effizient.

4. Hinzufügen eines neuen Kommentars:

- Füge ein neues Dokument in die `comments`-Sammlung ein.
- Optional: Inkrementiere `comment_count` im zugehörigen `posts`-Dokument. Dies kann atomar erfolgen, wenn die Datenbank es unterstützt, oder als separate Update-Operation (eventual consistency).
- Performance: Schnell. Die Aktualisierung des `comment_count` ist ein zusätzlicher Schreibvorgang, der aber die Leseleistung für die Anzeige der Kommentaranzahl verbessert.

Skalierbarkeit

- Horizontale Skalierbarkeit: Jede Sammlung (`users`, `posts`, `comments`) kann unabhängig voneinander auf verschiedene Shards verteilt werden.
- Dokumentgrösse: Durch die Referenzierung der Kommentare bleiben die `posts`-Dokumente relativ klein, was positiv für die Performance und die Vermeidung von Grössenlimits ist.
- Lese-Skalierbarkeit: Häufige Leseoperationen (Post anzeigen) profitieren von den Denormalisierungen und der Trennung der Sammlungen.
- Schreib-Skalierbarkeit: Schreibvorgänge sind ebenfalls gut skalierbar. Die Aktualisierung denormalisierter Daten (z.B. wenn sich ein username ändert) erfordert mehr Aufwand (Aktualisierung in `users` und in allen zugehörigen `posts` und `comments`). Dies ist ein bekannter Trade-off. Für Anwendungsfälle, in denen Benutzernamen sich selten ändern, ist dies oft akzeptabel. Wenn sie sich häufig ändern, könnte man erwägen, Benutzernamen nicht in Posts/Kommentaren zu denormalisieren und stattdessen bei der Anzeige separate Lookups durchzuführen oder die Aktualisierung über Hintergrundprozesse zu steuern.