**Discussion 3.1 – Normalization**

Patrick Cuauro

Bellevue University

WEB 335-307O Introduction to NoSQL

Professor Krasso

June 6, 2023

# What's data Normalization?

Normalization is the process of organizing data in a database. This includes creating tables and establishing relationships between those tables according to rules designed both to protect the data and to make the database more flexible by eliminating redundancy and inconsistent dependency.

# Why it is important?

It is important that a database is normalized to minimize redundancy (duplicate data) and to ensure only related data is stored in each table. It also prevents any issues stemming from database modifications such as insertions, deletions, and updates.

# Normalization forms

## First normal form

Eliminate repeating groups in individual tables.

Create a separate table for each set of related data.

Identify each set of related data with a primary key.

Do not use multiple fields in a single table to store similar data. For example, to track an inventory item that may come from two possible sources, an inventory record may contain fields for Vendor Code 1 and Vendor Code 2.

What happens when you add a third vendor? Adding a field is not the answer; it requires program and table modifications and does not smoothly accommodate a dynamic number of vendors. Instead, place all vendor information in a separate table called Vendors, then link inventory to vendors with an item number key, or vendors to inventory with a vendor code key.

## Second normal form

Create separate tables for sets of values that apply to multiple records.

Relate these tables with a foreign key.

Records should not depend on anything other than a table's primary key (a compound key, if necessary). For example, consider a customer's address in an accounting system. The address is needed by the Customers table, but also by the Orders, Shipping, Invoices, Accounts Receivable, and Collections tables. Instead of storing the customer's address as a separate entry in each of these tables, store it in one place, either in the Customers table or in a separate Addresses table.

## Third normal form

Eliminate fields that do not depend on the key.

Values in a record that are not part of that record's key do not belong in the table. In general, anytime the contents of a group of fields may apply to more than a single record in the table, consider placing those fields in a separate table.

For example, in an Employee Recruitment table, a candidate's university name and address may be included. But you need a complete list of universities for group mailings. If university information is stored in the Candidates table, there is no way to list universities with no current candidates. Create a separate Universities table and link it to the Candidates table with a university code key.

EXCEPTION: Adhering to the third normal form, while theoretically desirable, is not always practical. If you have a Customers table and you want to eliminate all possible interfield dependencies, you must create separate tables for cities, ZIP codes, sales representatives, customer classes, and any other factor that may be duplicated in multiple records. In theory, normalization is worth pursuing. However, many small tables may degrade performance or exceed open file and memory capacities.

It may be more feasible to apply third normal form only to data that changes frequently. If some dependent fields remain, design your application to require the user to verify all related fields when any one is changed.

## Other normalization forms

Fourth normal form, also called Boyce Codd Normal Form (BCNF), and fifth normal form do exist, but are rarely considered in practical design. Disregarding these rules may result in less than perfect database design, but should not affect functionality.

# How are 1-to-many relationships represented in MongoDB

The 1:N relationship can be modeled in several different ways using MongoDB. In this chapter we will explore three different ways of modeling the 1:N relationship. The first is embedding, the second is linking and the third is a bucketing strategy that is useful for cases like time series. Let’s use the model of a Blog Post and its Comments.

## Model

|  |
| --- |
| {  title: "An awesome blog",  url: "http://awesomeblog.com",  text: "This is an awesome blog we have just started"  } |

Example of a Blog post document

|  |
| --- |
| {  name: "Peter Critic",  created\_on: ISODate("2014-01-01T10:01:22Z"),  comment: "Awesome blog post"  }  {  name: "John Page",  created\_on: ISODate("2014-01-01T11:01:22Z"),  comment: "Not so awesome blog"  } |

Some example Comments documents

# Embedding

|  |
| --- |
| {  title: "An awesome blog",  url: "http://awesomeblog.com",  text: "This is an awesome blog we have just started",  comments: [{  name: "Peter Critic",  created\_on: ISODate("2014-01-01T10:01:22Z"),  comment: "Awesome blog post"  }, {  name: "John Page",  created\_on: ISODate("2014-01-01T11:01:22Z"),  comment: "Not so awesome blog"  }]  } |

A Blog Post with Embedded documents

The embedding of the comments in the Blog post means we can easily retrieve all the comments belong to a particular Blog post. Adding new comments is as simple as appending the new comment document to the end of the comments array.

However, there are three potential problems associated with this approach that one should be aware off.

The first is that the comments array might grow larger than the maximum document size of 16 MB.

The second aspects relates to write performance. As comments get added to Blog Post over time, it becomes hard for MongoDB to predict the correct document padding to apply when a new document is created. MongoDB would need to allocate new space for the growing document. In addition, it would have to copy the document to the new memory location and update all indexes. This could cause a lot more IO load and could impact overall write performance.

## Important

It’s important to note that this only matters for high write traffic and might not be a problem for smaller applications. What might not be acceptable for a high write volume application might be tolerable for an application with low write load.

The third problem is exposed when one tries to perform pagination of the comments. There is no way to limit the comments returned from the Blog Post using normal finds so we will have to retrieve the whole blog document and filter the comments in our application.

Sources:

<https://learn.microsoft.com/en-us/office/troubleshoot/access/database-normalization-description>

<https://towardsdatascience.com/database-normalization-explained-53e60a494495?gi=a0ead5921fd9#:~:text=Normalization%20is%20a%20technique%20for,insertions%2C%20deletions%2C%20and%20updates>.

<https://learnmongodbthehardway.com/schema/schemabasics/#:~:text=retrieve%20the%20document.-,One%2DTo%2DMany%20(1%3AN),related%20to%20a%20single%20Blog>.