Single Collection Pattern

- This pattern includes different kinds of documents in a single collection.
- It allows us to avoid multiple queries to read non-embedded related documents.
- It allows **$lookup** operations to be avoided.
- It is used for M:N relationships when we want to avoid data duplication and embedding isn't an option.
- It is also used to model 1:N relationships.
- Some real world use cases for this pattern are: A Catalog of Product Items, An Online Shopping Cart.
- This pattern has 2 variants: The first applies to M:N relationships and the second applies to 1:N relationships.
- The **first variant** uses an array of references and a documentType (docType) field that allows us to model M:N and 1:N relationships.
- In the **bookstore example** there are 3 collections: Reviews, Users and Books, one for each entity used to model the Book's catalog feature.
- So, all these documents should have a **docType** field which contains the type of document. This allows us to query the documents by type, as if they were in their own respective collections.
- After that, the **relatedTo** field is added. This will help model relationships between documents and the new collection using an array of references. All the documents in our collection are related to the same **bookId**. The Book document points to itself.
- Now, all the documents (book, user, review) to the books catalog collection.
- To complete the implementation of the single collection pattern, we have to create an index on the **relatedTo** array to support our application queries. This is done by using: **db.books_catalog.createIndex({relatedTo: 1}).**
- We can retrieve all info related to the books catalog collection using: **db.books_catalog.find({"relatedTo": 202356})**.
- We can also do queries like: **db.books_catalog.find({"relatedTo": 202356, "docType": "review"})**.
- The **second variant** of the single collection pattern uses an overloaded field. A field is overloaded when it is used for a purpose other than its original intent.

- In this case, we're using the query to identify documents in the single collection and also to enable efficient queries. This variant is used to model 1:N relationships only.

- Let's look at a pattern to model a **1:N** relationship.

- To do this, the single collection id field is added and overloaded (**sc_id**).

- The Book document uses the **book_id** and the **sc_id**. The Review uses an overloaded **sc_id** that contains the book_id and sc_id values in that order separated by a slash.

- The Books document should be moved to the book_catalog **collection**.

- The currently created Book and Review documents should be moved to the books_catalog collection.

- An index needs to be created for the collection using: **db.books_catalog.create_index({sc_id: 1})**.

- Some examples of queries to the book catalog collection are listed below:

- **db.books_catalog.find({"sc_id": {"$regex": "^202356"}})** (Returns all book documents starting with the given book_id).

- **db.books_catalog.find({"sc_id": {"$regex": "^202356/"}})** (Returns only the reviews of the book with the given book_id).

Subset Pattern:

- Some of the books in the Bookstore App have thousands of reviews. All these reviews take up too much memory. The App displays only 3 reviews per book.

- To efficiently run queries, **WiredTiger**, MongoDB's database engine keeps data that is frequently accessed together in memory. This in-memory set is called the **WiredTiger internal cache**.

- Ideally, the **Working Set**, i.e., the portion of the indexes and documents frequently used by the application should fit into **WiredTiger's internal cache**. DB performance is impacted when the working set exceeds the internal cache size.

- The **Subset Pattern** can be used to mitigate these problems by reducing the document size. It does this by relocating data that isn't frequently accessed. When more documents can be fitted into the internal cache, then DB performance increases.

- The Subset Pattern is used when we have a document with a large number of embedded subdocuments such as reviews or comments stored in an array, but only a small subset of those subdocuments are regularly used by our application.

- **An example is described here**. In the application, we have a Book document that has thousands of review documents in it. However, only 3 of those review documents are being accessed frequently. In this situation, we should create a separate reviews collection and put all the reviews in the book document over there. We should copy the first 3 reviews from the reviews collection and put them in the book document.

- The above measures reduce the size of the Book document. Smaller documents also reduce cache usage and increase query performance. This also helps keep data that is accessed together, stored together.

- We're going to apply the Subset pattern by running 2 data pipelines on the Books collection using MongoDB's Aggregation Framework.

- The first pipeline will copy all reviews from all the Book documents in the Books collection and store those reviews as separate Review documents in the Reviews collection.

- After the above step, the second pipeline will ensure that the Review array in each Book document contains no more than 3 reviews.

- The **first pipeline** will use the **$unwind** stage to deconstruct the reviews array field and create a new document for each review. However, each of these review documents will contain all the data from the corresponding original Book document. Since we need to use the Product ID to associate each Review document with its corresponding Book document, we will use the **$set** stage to add the Product ID to each Review document. Then, we will use the **$replaceRoot** stage to promote the embedded Review document to the top level. Now, each of the review documents will contain review data and a Product ID. Finally, we will take the **$out** stage to take all of the Review documents and write them to a new Reviews collection. If a Reviews collection already exists, then the **$out** stage will overwrite it with the output of the pipeline.

- The **second pipeline** will use the **$set** stage to overwrite the existing reviews field. We will use the **$slice** operator to keep the first 3 reviews from the reviews array.

Bucket Pattern:

- Suppose, we want to record the number of views per book to analyze customer viewing patterns, then we would want to model the relationship between a book and its views as 1:N. However,we expect the number of user views to be very large in some cases. Embedding isn't a good solution as it will lead to very large document sizes. Referencing isn't good either as it will decrease performance due to large indexes and very complex queries.

- The Bucket Pattern helps us group information into buckets so that the document size becomes more predictable for our system.

- This pattern is often used with IoT sensors to handle their readings.

- This pattern is used with the pre-computed pattern to store pre-computed statistics in the buckets. This helps improve performance further.

- **The Bucket Pattern When Used Properly, Can Do The Following For Us:**
- Keep document size predictable.
- Read only the data that we need.
- Reduce the number of documents in a collection.
- Improve Index Performance.

- When a user reads one of our Books, we want to capture the book_id, the timestamps and the user_id. We must also decide how the data will be queried so that we can understand how granular the buckets will be.

- Our most important queries need to compute monthly values so we will group monthly views per book in our bucket and store them in our Views collection.

- In the bucket, we should create a sub-document called **id** which will contain the **book_id** and **month**. The bucket will also contain an array called **views** which will store in respective sub_documents, the **timestamp** and **user_id** of each view.

- The above array shouldn't be unbounded. To prevent that, a schema validation tool should be used. Another way to ensure no unboundedness is to apply a threshold on the number of views in each bucket through logic in our application.

- If the array in a bucket exceeds its size, then the application will create another bucket. This will help keep our buckets to a predictable size.

Outlier Pattern:
- The Outlier Pattern helps us to deal with edge cases in an application.

- Imagine there's a feature in the Bookstore App that helps users put comments in reviews and that the **comments** KV pair is embedded as an array in each **Review** document.

- If a controversial book's review gets millions of comments, then the **comments** array could be unbounded and the document may reach the 16MB size limit.

- To prevent this, we can use the Outlier Pattern which allows us to treat documents with unusual characteristics differently from normal documents. This pattern allows us to handle these documents without redesigning the application as doing so, would degrade performance.

- The pattern is implemented by marking exceptional documents with an **outlier field**.

- First we set a threshold of 3 comments. If any Review has more than 3 comments, then it will be marked with an **outlier field**. The **outlier field** won't change the size of the **comments** array.

- The moment that review receives more than 3 comments, then those comments will be moved by the app.

- A separate query can also be run to move the excess comments to another collection.

- Another solution would be to use the Bucket Pattern to store excess comments in buckets.

https://learn.mongodb.com/learn/course/advanced-schema-design-patterns/lesson-1-single-collection-pattern/learn?client=customer