NoSQL Implementation With CosmosDB API For MongoDB

Peter L. Clayton, Computer Science

CPSC-652, University of Bridgeport

ABSTRACT

The purpose of the project is to highlight NoSql features. The project implements a website that connects to a NoSQL database and serves the results back to the website user. The project entitled StudentPerformanceApp represents the start of a user management system to add person's affiliated with a school such as students, teachers, TAs, lecturers, guests etc.

1. INTRODUCTION

The StudentPerformance APP implements the NoSQL Document model based on the CosmosDB API for MongoDB. The project is implemented as a Blazor website hosted on the azure platform which connects to a separate API service that controls access to the database. The resulting API calls are presented to the user in a tabular format indicating success or failure of the operation. The application architecture allows for easy extensibility to provide more complex operations.

This paper will cover the chosen document model, Cosmos DB API for MongoDB, the application architecture, relevant NoSQL operations utilized in the project, and a screenshot of the application.

2. DOCUMENT MODEL

The Document Model for the Student Performance App allows for easy insertion of "documents" (similar to records in a SQL Table) into "collections" (similar to Tables in SQL). The power of the document model is that collections can contain uneven shapes of documents. Each document is allowed to contain its own unique number of key-value pairs that contain data. Additionally, each value can itself be its own key-value pair which allows for nesting of data arrays and complex search queries not possible in a relational database model.

2.1. Multiple Schema Options

The benefits of the Document Model provide for use cases that can be schema less, semi structured, or fully structured offering a significant number of options for database architects.

2.2 Sharding & Big Data

The Document Model as implemented by MongoDB enables horizontal scaling of compute resources to manage the access to and processing of data. The advantage is that many computers break up large workloads into smaller, faster, and more efficient operations that allow the system to handle incredibly large volumes of data.[1]

2.3 Aggregation

While MongoDB does have support for MapReduce, it provides a more framework efficient series of functions for statistical analysis and data management called aggregators. Aggregation is the preferred way to run calculations across massive and complex data sets within the MongoDB world.

3. COSMOS API FOR MONGODB

3.1. CosmosDB Platform

The Selected Database implementation is the Azure CosmosDB API for MongoDB. The Azure platform is a cloud service by Microsoft which hosts a variety of services and applications, of which Cosmos DB implements Azure's database functionality. Cosmos DB allows access to several backend engines including their own SQL, Cassandra, Gremlin, and MongoDB implementation.

CosmosDB provides access via its API, a provided MongoDB Shell, terminal/command line options, and via its portal interface.

The fully managed cloud architecture of CosmosDB provides tools and resources for monitoring and administering its related database frameworks. The key benefit is that the natural power of MongoDB to handle large data volumes can be realized in practical, real world

enterprise settings. While MongoDB databases have the ability to scale, CosmosDB provides the actual structure instantly at low cost and with high availability. Shards, and replications not only happen among systems and servers, but can happen across globally located data centers. Furthermore, CosmosDB allows for the elastic scale up and scale down of resources to meet resource demand, reduce administrative and capital costs. These services can all be setup to happen automatically providing even more resource leveraging efficiencies.

3.2. CosmosDB API (for MongoDB)

Cosmos DB API for MongoDB accesses its custom MongoDB database engine with its own methods, classes, and interfaces. These methods closely resemble typical MongoDB functions such as "Find", "InsertOne", "Set", and other common naming conventions. Additionally, comfort with the BSON (Binary JSON) formatting approach is important for creating the appropriate filters and queries.

4. APPLICATION ARCHITECTURE

The StudentPerformanceAPP architecture contains a front end composed with Blazor that connects to a database microservice via HTTPS which then connects to CosmosDB with MongoDB method queries and posts.

Blazor Front End: The client application separates CRUD based operations on 5 pages: List All, Create, Read, Update, and Delete pages, that highlight each respective functionality. Use of the Blazor component model allows for front-end development with C# and the creation of reusable custom code blocks such as the tables used for displaying the data

API Back End: StudentPerformanceAPP implements a layered approach for the service. The controller page acts as the business layer, which sends data to the repository layer, and finally reaches out to the data access layer. This allows for loose coupling and the ability to switch out database implementations with minimal code refactoring

The Data Access Layer provides the MongoDB functionality with method calls and connections to the database service. For the CosmosDB API for MongoDB, several of the methods require formatting the query and filter parameters for submission. For example, when performing an update on a field or set of fields it's important to use builder functions (defined in the API) to create FilterDefinitions and UpdateDefinitions object types in order to process the set commands used in updating. It was therefore important to understand the appropriate MongoDB syntax as well its translation into the API's custom usage.

5. NOSQL OPERATIONS

```
GetCollection<T>(container).Find
GetCollection<T>(container).Find(filter).Limit(50).ToListAsync<T>();
GetCollection<T>(container).InsertOneAsync(document);
GetCollection<T>(container).UpdateManyAsync(bsonFilter,
bsonNewData);
GetCollection<T>(container).DeleteOneAsync(bsonQuery);
```

NoSQL implementation is the key requirement for this project. The above listed methods from the project source code are the standard MongoDB document operations that allow for finding, inserting, updating, and deleting. The powerful aspects of these methods are the filter, query, and custom BSON documents method arguments that can represent highly efficient data retrieval and manipulation that would require impractical resources for a highly complex, large volume system hosted in an RBDMS with expensive join operations.

11. CONCLUSION

The Student Performance App highlights the usage of a cloud hosted service for a MongoDB NoSQL implementation by using flexible architecture design, fast and efficient methods to provide data.

The significance of this project lies in the easy ability to extend its capabilities to make use of complex queries, use of the aggregation functions of MongoDB, and it's CosmosDB implementation which allows for global replication at scale without code modification, managed sharding of data for low latency, and efficient access, and high availability at scale.

A future looking extension of this project should support nested document access and display. As the project's design is layered and loosely coupled from front-end to back-end, adding support would be quite feasible and very powerful

11. REFERENCES

[1] https://docs.mongodb.com/manual/sharding/

[2] project website:
https://studentperformanceappclient.azurewebsites.net/

[3] project api url:
https://studentperformanceapideploy.azurewebsites.net/

[4] github:
https://github.com/peterlloydclayton/UB-StudentPerformanceApp