nucoris

Persistence

Jaume Canals

Spring 2019

Contents

1 Introduction

nucoris is the exploratory prototype of a cloud-native healthcare information system (HIS) that I have developed and used as a learning platform during a sabbatical leave.

As most business applications, an HIS handles a significant amount of data that must be stored somewhere. A relational database is the type of storage most commonly used, but some capabilities of cloud NoSQL databases make them an interesting alternative to consider.

This document provides an overview of Azure Cosmos DB features and restrictions, and explains how I have designed **nucoris** persistence layer to overcome Cosmos DB limitations while ensuring the best performance for the lowest price possible.

I conclude the document with my impressions on the type of applications that are a good fit for Cosmos DB.

2 Overview of Azure Cosmos DB

Azure Cosmos DB is Microsoft's multi-model NoSQL database cloud service. Let's review what this means:

2.1 Main Features

2.1.1 Multi-Model

You can store and access data with **different APIs and formats**: MongoDB or SQL for a document database, Gremlin for a graph database and Table API for key-value storage.

Cosmos DB is **schema-agnostic** and **automatically indexes** all the data without explicit management from developers.

It supports stored procedures, triggers and user-defined functions written in JavaScript.

2.1.2 Global Distribution

Cosmos DB provides the ability to globally distribute data across all Azure regions and serve the data locally.

That is, you can choose to deploy your Cosmos database to Western Europe and East Australia, for example, and Azure takes care of **replicating changes between the two regions automatically in near real time** (with the unavoidable delay introduced by the distance). Azure will serve the data from the region closest to the calling user/application.

2.1.3 High Availability and Scalability

Cosmos DB guarantees **less than 10-ms latency 99.99% of the time**, for both reads and writes, all around the world.

It guarantees as well **99.99% availability within a region** and 99.999% availability across multiple regions. That is, at most one hour and 5 min of downtime throughout the year, respectively.

2.1.4 Change Feed

Change Feed is a Cosmos DB feature that works by listening to changes in a Cosmos DB container. It then asynchronously outputs (to an Azure Function, for example) the documents that were changed, in the order in which they were modified.

You can use this interesting feature to trigger additional actions such as sending documents to an Azure Service Bus or to maintain a data warehouse in sync with Cosmos DB, in near real time.

2.2 Databases and Collections

In a document DB, **documents are stored in collections**. Each database has at least one **collection**. Note that you may be tempted to assimilate collections to relational tables, but they are quite different concepts: a collection can hold an unlimited amount of objects (documents) of any type (schema).

It's quite common that an application is designed to use a single collection, but several Cosmos DB properties are scoped at the collection level and may prompt you to deploy more than one collection in your database. Keep in mind the following facts:

1) Cosmos DB is priced at *request units per second* (RU/s). Reading a 1 kbyte document is supposed to cost 1 RU. Microsoft currently forces you to provision at least 400 RU/s per collection, so the more collections you have, the more money you have to pay.
2) Every item in a collection must have a property that you will define as its partition key. The property is specified as a JSON path of the documents to be stored in the collection, and it's unique per collection.
   If you are unable to specify the same property as partition key for all your items you may need to create different collections.
   A common workaround is to explicitly add to every item a "partitionKey" property whose contents are controlled by your application, and to define then the JSON path of the collection's partition key as "/partitionKey".
3) Each collection has also an indexing policy that applies to all items of the collection. By default all properties are indexed. This is convenient in queries but causes higher RU/s

when writing. If you want to fine tune the indexing policy and you have items with very different shapes you might prefer to create different collections.

2.3 Partitions

Partitioning is the technique that allows Cosmos DB to horizontally scale individual containers in a database to meet the performance and availability needs of applications. Items are divided into subsets called *logical partitions*, based on the value of the *partition key* property associated with each item, which is the same for all items in a collection. Each item must have a partition key and an id (id must be unique within a logical partition).

Choosing the partition key is an important decision that affects the application's performance, your ability to efficiently query the database, and how you persist your changes into a database. In general, your goal is to spread uniformly the data and the activity in your container across the set of logical partitions, to avoid hot spots.

2.4 Transactions

Cosmos DB **supports transactions but in a limited way**:

- Only statements executed within a stored procedure are transactional.
- A stored procedure belongs to a single collection, so a transaction cannot span more than one collection.
- Moreover, execution of stored procedures is scoped at the partition level, so a transaction cannot span more than one partition in a collection.

The only good news is that you don't need to take additional steps: all statements in a stored procedure are automatically enlisted in a transaction behind the scenes, no need to call any *BeginTransaction* method.

2.5 Queries

You can query Cosmos DB database with a **SQL API**, either in Azure Cosmos DB portal or with a .NET SDK (available as a NuGet package for .NET and .NET Core).

Although quite powerful, it's significantly more limited than relational SQL:

1) You cannot query across different collections, only within a collection.
2) Queries can span several partitions but it's not recommended because it increases the RU/s. You should strive to query within a partition.
3) You can't join documents within a collection, only nested objects within the same document.

2.6 Pricing

Azure has a price estimate calculator you can use to estimate how much your Cosmos DB may cost. You pay along two dimensions: **storage** and **provisioned throughput**.

You have to provision at least 400 RU/s, which it's currently priced at $0.008/h ($23.36/month).

Storage is priced at $0.250/GB/month. So, for a 100 GB DB you would pay $25/month.

If your RU/s are low, you may get cheaper prices than with an Azure SQL database.

2.7 Cosmos DB Emulator

For local development and testing Microsoft provides a very convenient Azure Cosmos DB Emulator, which can run also on Docker for Windows.

3 Data Modeling in Cosmos DB

3.1 Constraints

You've probably heard that **document databases are "schema-less" and let you store objects (JSON documents) of any shape**. Some might infer from this fact that this simplifies storing the data. My experience is exactly the opposite: **designing the persistence model for Cosmos DB requires more thought than for a relational database**. The reasons are:

1) You have to take into account the Cosmos DB pricing model in your design:
    a. The more collections you define, the higher the minimum price (because of the 400 RU/s low limit you have to pay per collection).
    b. The larger the documents, the more you pay.
    c. The more document properties you index to enable queries, the more you will eventually have to pay because of higher RU/s.
2) Performance is strongly influenced by your choice of partition key, which is unique per collection:
    a. In writes, you have to avoid hot-spots by distributing them across different partition keys.
    b. In queries, you have to avoid queries spanning several partition keys because they are more expensive in terms of RU/s (and more RU/s implies more money).
3) Document DBs are optimized to host self-contained documents, but most domain models express rich relationships that you will have to transform into JSON documents (limited to 2 MB, by the way). There are some well-known patterns, but they all require careful work and tradeoffs.
4) Transactions can only span a single partition key of a single collection. Either you design your persistence model to fit into this restriction or you develop extra infrastructure to react to occasional failures that might leave your data in an inconsistent state.
5) Although Cosmos DB can index every property of a document and provides a SQL API to query the document database, there are significant query limitations compared to a relational database. When designing the persistence model you need a clear understanding of the queries your application will run, because:
    a. You cannot write a query spanning two collections.
    b. You cannot write conditions that join different documents in a collections: joins are limited to properties within a document.
    c. Queries across partition key boundaries are more expensive and should be avoided.

d. Indexing requires more RU/s at write time, which means more money. So, the more properties you index to enable your queries, the more you pay.

3.2 Nucoris Solution

In this section I explain the decisions I've made in **nucoris**.

3.2.1 Single Database & Single Collection

Having more than one collection costs more money and has no performance or maintenance advantages, so we have decided to have **a single database with a single collection**.

3.2.2 Partitioning by Data Category

In **nucoris** we have identified the following data categories:

1) **Patient** Data: data associated to a patient: demographic information, allergies, medications prescribed, laboratory analysis results, collected vital signs, etc.
2) **User** Data: data associated to a system user. As you would expect it includes full name, username, authorizations, roles, etc., but for clinical users it includes as well a description of its clinical competences, accreditations, shifts, etc.
3) **Reference** Data: dictionary data such as list of medications, list of units, list of departments, etc. Every item in this category has at least a property with its name.
4) **Materialized View** Data: the previous categories are common to any clinical application, but "materialized view data" is an artificial category that I have introduced to overcome the querying limitations of Cosmos DB, in particular the advice to limit the queries to a single partition. We'll explain it in more detail further below, for now it suffices to say that this category groups ad-hoc items defined to populate UI worklists that contain data of more than one patient.

Remember that for every collection in a Cosmos database you have to designate **a single property of every collection item as the partition key** of the collection, specifying it as a JSON path. Typical easy examples you may find in Internet are "/customerId" for collections storing customer data, or "/client/address/city" for collections storing order data.

If you store unrelated items in the same collection, as we do in **nucoris**, there is no property you can naturally come up with as partition key. You have to **add an artificial property to every item, which here we call "partitionKey" for clarity**, and define the JSON path as "/partitionKey".

But once we've made this decision, to which value we should set partitionKey? The main criteria that should drive assignment of partition keys are:

1) Ensure a good distribution of values, to avoid hot spots. In the end, data is stored in physical SSD disks, and Cosmos DB will try to store different partitions in different

disks to improve its overall performance, so you do not want your application to write data to the same disk (partition) all the time.

2) Avoid queries spanning more than one partition. Since different partitions may be stored in different disks, you do not want to run a search not filtered by partition, because it would require searching in all the disks at the same time.
3) Aim for transactional persistence of related items. Since transactions are scoped per partition key, you should strive to write items changed by the same command into the same transaction/partition.

With these goals in mind, this is how nucoris assigns partition key values for each category:

1) **Patient Data**: since we'll often search for items of a particular patient, for patient data we've chosen patientId as partition key, in the format: **"P_" + patientId**. The prefix "P_" is introduced for clarity if manually inspecting the documents and to ensure there will not be collisions with other categories. Given that patientId is a GUID, these are some examples of partition key values in this category:
   P_8a43bd02-8c4d-4841-b7d8-cc1c37c4811c
   P_3141b415-5f31-47c8-9165-74bed413a434

2) **User Data**: since we'll often search for details of a particular user, for user data we choose userId as partition key, in the format: **"U_" + userId**. Given that userId is a GUID, these are some examples of partition key values in this category:
   U_a247196b-4f1d-4e2a-9fa7-f80432706e7f
   U_c63b8298-7f18-4318-965b-d6ee3c86f769

3) **Reference Data**: since we'll often query dictionary data to get all items of a given type or a specific item of a given type (e.g. all medications, a specific department), as partition key we choose **"R_" + type name**. Examples:
   R_Medication
   R_Department

4) **Materialized View Data**: in the user interface of the application we will have worklists such as "Admitted Patients" or "My Patients" that list one row for each patient that fulfills the worklist's filtering criteria. If we relied on patient data stored in a distinct partition per patient ("P_" + patientId), populating such worklists would require a query on all patient partitions. This is not efficient and should be avoided.
   The only solution is to have **a partition per UI worklist**, containing an item for every candidate row in the worklist (in a subsequent section I explain how such partitions are maintained). This is akin to **materialized views** or **projections in the CQRS pattern**, since they are built by projecting selected properties of multiple patients into items

that are then persisted.
Examples of partitions that hold query data:

a. We can define a partition containing an item for every patient, with properties such as demographic data and admission state. Its partition key will be:
V_PatientStateView
b. We can define a partition to hold an item for each active order, with properties such as order description, order state, patient id, assigned user id, etc. Its partition key will be:
V_ActiveOrdersView

In summary:

Data Category	Partition Key Pattern	Example of partitionKey Value
Patient	P_ + Patient Id	`P_3141b415-5f31-47c8-9165-74bed413a434`
User	U_ + User Id	`U_c63b8298-7f18-4318-965b-d6ee3c86f769`
Reference	R_ + Type of data	`R_Medication`
Materialized View	V_ + View name	`V_PatientStateView`

Note: we have also introduced a reference partition R_User to keep the basic data of all users; it's useful when you have to show a list of users.

3.2.3 Storing Patient Data

A rich domain model has always parent-child relationships. In the clinical domain, many different data types can be associated to a patient: demographic data, allergies, medications prescribed, assessments ordered, tests performed, vital signs collected from devices, etc. They could all be modeled as children or descendants of a patient entity.

Storing all of them into a single patient document doesn't make sense: it would exceed Cosmos DB document size limit and it would be terribly inefficient to update. We need more granular documents.

What **nucoris** does instead for patient data is **to store:**

- **A document per *aggregate root* in the patient's domain model**
  - Note: children of aggregate roots that are also aggregate roots themselves (e.g. admissions of a patient, orders of an admission) are each serialized as a separate document, not as child properties of the parent document.
- **A document for each event raised**.

So, in a patient's partition you will find:

1) Exactly one document with the patient demographic data (domain class `Patient`)
2) A document for each patient's admission (domain class `Admission`).
3) A document for each order (`MedicationPrescription`, `RadiologyExam`, etc.)
4) A document for each domain event raised when executing a command on patient data (`PatientCreatedEvent`, `AdmissionStartedEvent`, `MedicationPrescribedEvent`, etc.)

This design supports **efficient updates and queries** while maintaining a **detailed audit trail of all actions performed on patients**.

3.2.4 Document Layout

A document database stores JSON documents, which in our case are serialized .NET objects. In Cosmos DB, each document must have an "**id**" property (if not present, Cosmos DB would automatically add it and set it to a GUID). As discussed above, our documents have also a "**partitionKey**" property.

Additionally, it's often interesting to have metadata properties to filter items in a partition by its conceptual type (for example, to query all active orders of a patient). To this effect we have introduced a property called "**docType**".

Finally, in some scenarios where the domain has defined a hierarchy of subtypes (e.g. an abstract class with several derived classes) we may have to fetch a collection of documents of different types and deserialize them without knowing their exact .NET type beforehand. In such scenarios is definitely useful to know their .NET type full name, which we store in the property "**docSubType**".

So, we have <u>4 metadata properties we want to store together with our domain entities</u>: *id*, *partitionKey*, *docType* and *docSubType*.

The easiest way to persist them would be to define these properties as members of the domain base class `Entity` or `AggregateRoot`, but this approach would pollute the domain model with persistence concerns to a degree I consider excessive.

Instead what I've chosen is to wrap at write/read time the persistable domain objects in a `DbDocument` class that has as members those 4 properties plus the domain object itself (in a property named "**docContents**"):

```
public class DbDocument<T>
{
    public string id { get; set; }
    public string partitionKey { get; set; }
    public string docType { get; set; }
    public string docSubType { get; set; }
    public T docContents { get; set; }
}
```

With this approach, the JSON documents look like this in Cosmos DB:

3.2.4.1 Example of a Patient JSON Document

```
{
    "id": "c63b8298-7f18-4318-965b-d6ee3c86f769",
    "partitionKey": "P_c63b8298-7f18-4318-965b-d6ee3c86f769",
    "docType": "Patient",
    "docSubType": "nucoris.domain.Patient",
    "docContents": {
        "$type": "nucoris.domain.Patient, nucoris.domain",
        "HasSystemGeneratedMrn": false,
        "GivenName": "GN_557db63f-a631-499f-b7b8-9abdb2475ef1",
        "FamilyName": null,
        "DateOfBirth": "2019-02-21T08:35:41.5469058Z",
        "Allergies": [],
        "PatientIdentity": {
            "Id": "c63b8298-7f18-4318-965b-d6ee3c86f769",
            "Mrn": "MRN_ActiveOrdersQueryTests_CRUDOperations_WorkAsExpected_557db63f-a631-499f-b7b8-
9abdb2475ef1"
        },
        "Id": "c63b8298-7f18-4318-965b-d6ee3c86f769"
    },
    "_rid": "ehM1AOZ+-EWpCwAAAAAAAA==",
    "_self": "dbs/ehM1AA==/colls/ehM1AOZ+-EU=/docs/ehM1AOZ+-EWpCwAAAAAAAA==/",
    "_etag": "\"00000000-0000-0000-c9c0-6df3830501d4\"",
    "_ts": 1550738141
}
```

3.2.4.2 Example of a Domain Event JSON Document

Example of the domain event raised when that patient was created, which is stored in the same patient partition because it's patient data too:

```
{

    "id": "863ec4a5-d610-4275-9e10-b60b3c9184ce",
    "partitionKey": "P_c63b8298-7f18-4318-965b-d6ee3c86f769",
    "docType": "Event",
    "docSubType": "nucoris.domain.PatientCreatedEvent",
    "docContents": {
        "$type": "nucoris.domain.PatientCreatedEvent, nucoris.domain",
        "Description": "Patient created",
        "Patient": {
            "HasSystemGeneratedMrn": false,
            "GivenName": "GN_557db63f-a631-499f-b7b8-9abdb2475ef1",
            "FamilyName": null,
            "DateOfBirth": "2019-02-21T08:35:41.5469058Z",
            "Allergies": [],
            "PatientIdentity": {
                "Id": "c63b8298-7f18-4318-965b-d6ee3c86f769",
                "Mrn": "MRN_ActiveOrdersQueryTests_CRUDOperations_WorkAsExpected_557db63f-a631-499f-b7b8-
9abdb2475ef1"
            },
            "Id": "c63b8298-7f18-4318-965b-d6ee3c86f769"
        },
        "TriggeredBy": null,
        "EventTime": "2019-02-21T08:35:41.5513795Z",
        "PatientIdentity": {
            "Id": "c63b8298-7f18-4318-965b-d6ee3c86f769",
            "Mrn": "MRN_ActiveOrdersQueryTests_CRUDOperations_WorkAsExpected_557db63f-a631-499f-b7b8-
9abdb2475ef1"
        },
        "Id": "863ec4a5-d610-4275-9e10-b60b3c9184ce"
    },
    "_rid": "ehM1AOZ+-EWqCwAAAAAAAA==",
    "_self": "dbs/ehM1AA==/colls/ehM1AOZ+-EU=/docs/ehM1AOZ+-EWqCwAAAAAAAA==/",
    "_etag": "\"00000000-0000-0000-c9c0-6df3a3a001d4\"",
    "_ts": 1550738141
}
```

3.2.4.3 Example of a Materialized View JSON Document

Example of an item created in the materialized view partition V_PatientStateView for the same patient to populate the *Admitted Patients* UI worklist and to support searches by patient's MRN and name (note the uppercase versions of MRN and name):

```
{
    "id": "c63b8298-7f18-4318-965b-d6ee3c86f769",
    "partitionKey": "V_PatientStateView",
    "docType": "View",
    "docSubType": "nucoris.queries.PatientStateQuery.PatientStateQueryItem",
    "docContents": {
        "$type": "nucoris.queries.PatientStateQuery.PatientStateQueryItem, nucoris.queries",
        "QueryPatient": {
            "Id": "c63b8298-7f18-4318-965b-d6ee3c86f769",
            "Mrn": "MRN_ActiveOrdersQueryTests_CRUDOperations_WorkAsExpected_557db63f-a631-499f-b7b8-
9abdb2475ef1",
            "GivenName": "GN_557db63f-a631-499f-b7b8-9abdb2475ef1",
            "FamilyName": null,
            "DateOfBirth": "2019-02-21T08:35:41.5469058Z",
            "State": "Discharged",
            "Admissions": [
                {
                    "Id": "72e58b11-adb7-45b8-8036-712f106c3ace",
                    "Started": "2019-02-21T08:35:41.6302006Z",
                    "Ended": "2019-02-21T08:35:42.6513176Z",
                    "IsActive": false
                }
            ],
            "UpperMrn": "MRN_ACTIVEORDERSQUERYTESTS_CRUDOPERATIONS_WORKASEXPECTED_557DB63F-A631-499F-B7B8-
9ABDB2475EF1",
            "UpperGivenName": "GN_557DB63F-A631-499F-B7B8-9ABDB2475EF1",
            "UpperFamilyName": null
        },
        "Id": "c63b8298-7f18-4318-965b-d6ee3c86f769"
    },
    "_rid": "ehM1AOZ+-EWrCwAAAAAAAA==",
    "_self": "dbs/ehM1AA==/colls/ehM1AOZ+-EU=/docs/ehM1AOZ+-EWrCwAAAAAAAA==/",
    "_etag": "\"00000000-0000-0000-c9c0-6ebb2cde01d4\"",
    "_attachments": "attachments/",
    "_ts": 1550738142
}
```

4 Maintaining the Items in Materialized Views

4.1 Review of Materialized View Partitions

I've discussed in previous sections the importance of ***materialized view partitions***, partitions that hold items designed to **support generic searches and to populate UI worklists containing rows from different patients**. This is an extract of the contents of one such view item, which contains data on a patient and its state (you can see the full document in the previous section):

```
"QueryPatient": {
            "Id": "c63b8298-7f18-4318-965b-d6ee3c86f769",
            "Mrn": "ab32556699",
            "GivenName": "John",
            "FamilyName": “Smith”,
            "DateOfBirth": "2019-02-21T08:35:41.5469058Z",
            "State": "Active",
            "Admissions": [
                {
                    "Id": "72e58b11-adb7-45b8-8036-712f106c3ace",
                    "Started": "2019-02-21T08:35:41.6302006Z",
                    "Ended": null,
                    "IsActive": false
                }
            ],
            "UpperMrn": "AB32556699",
            "UpperGivenName": "JOHN",
            "UpperFamilyName": “SMITH”
        }
```

Note that it contains **properties that can be displayed** directly (*Mrn*, *GivenName*, *FamilyName*) and **properties that can be used by searches** (*State* and uppercase versions of *Mrn*, *GivenName* and *FamilyName*, to facilitate efficient queries).

This solution is an effective workaround for the constraints Cosmos DB imposes on cross-partition queries. But how and when are such items created when a command handler changes data of a patient? There are at least two ways that we discuss below.

4.2 Synchronous Maintenance with Event Handlers

In **nucoris** application layer you can implement handlers of events raised by the domain entities. Such event handlers are implicitly invoked by `CommandSession` when command handlers finish their work and call session's `CommitAsync()` method.

You can create an event handler that listens to certain domain events that you know imply updates in the items of a certain materialized view partition. For example, the materialized view partition *V_PatientStateView* is composed of one item per patient, with the core patient identification data, current admission state and list of admissions. So, events

`PatientCreatedEvent`, `AdmissionStartedEvent` and `AdmissionEndedEvent` may imply inserting new items into this partition, or updates to existing items.

An event handler listening to these three events can extract the relevant data from the event and insert a new item, or update an existing item in the mentioned query partition (always through the appropriate repository injected into the event handler).

To demonstrate this approach I have implemented `PatientStatusChangeEventHandler` in the `nucoris.application` project, which relies on classes defined in the `nucoris.queries` project (folder `PatientState`).

This approach is quite straightforward to implement, and maintains the domain/application knowledge exactly where it belongs, but it has a **drawback: it increases execution time of commands**, and hence worsens the performance perceived by users, because changes done in the domain entities by command handlers are not committed until all event handlers finish.

4.3 Asynchronous Maintenance with Cosmos Change Feed

4.3.1 Overview

An alternative that does not affect runtime execution of commands is to use **Cosmos DB *Change Feed* feature**:

1) You deploy an *Azure Function* to listen to the collection's change feed.
2) The function sends the documents representing events (they have *docType="Event"*) into an *Azure Service Bus topic* we've labelled as "application", and ignores other document types.
3) Once events are injected into the *Application Bus*, you have several ways to process them to maintain the materialized view partition, depending on the degree of flexibility and encapsulation you want to achieve. I've considered these options, from more flexible to more encapsulated:
    a. You can deploy an Azure Logic App or Azure Function that subscribes to the "application" service bus topic, filters the events of interest, generates documents of the appropriate type and directly inserts/updates them into Cosmos DB with the appropriate partition key.
    b. You can deploy an Azure Logic App or Azure Function that subscribes to the "application" service bus topic, and posts event messages to a Web API that you create to maintain the materialized view partition.
    c. You can host a singleton service in the Web App that listens to messages in the "application" bus and invokes the application layer event handlers.

4.3.2 Application Event Loopback

The approach I've chosen in **nucoris** is what I call the **application event loopback**. In a nutshell, domain events are sent back to the application, where their handlers have access to the same rich infrastructure (domain entities, repositories, mediator, etc.) as "normal" synchronous event handlers.

In more detail, to maintain the `V_ActiveOrdersView` materialized view I've done the following:

1) I've deployed an Azure Function that has a *Service Bus Topic* as trigger, subscribing to the "application" bus where Cosmos DB change feed function sends event documents to.
2) The function uses an http call to post the JSON-serialized event document to a Web API called "ApplicationEvents".
3) The Web API "ApplicationEvents" is hosted by **nucoris** Web App and so its implementation has access to the same dependencies as regular application code (repositories, handlers, unit of work, etc.). The POST method extracts the domain event from the document and uses the injected mediator to publish the event.
4) I've implemented an event handler bound to *order events*, which inspects the incoming order event, queries the database to check for existing items in the `V_ActiveOrdersView` materialized view partition, and updates it as needed.

However, we can't actually bind the event handler directly to the domain event, because if we did it would be called also when the original command that raised the event was being processed. To prevent it we have introduced a wrapper class called `DeferredDomainEvent`, and we bind the loopback event handler to it:

```
public class DeferredDomainEvent<T> : MediatR.INotification where T : DomainEvent
{
    public T Event { get; }

    public DeferredDomainEvent(T domainEvent)
    {
        this.Event = domainEvent;
    }
}
```

A handler then looks like this:

```
public class ActiveOrderChangesEventHandler :
   MediatR.INotificationHandler<DeferredDomainEvent<MedicationPrescribedEvent>>,
```

5 Indexing Policy

The clean separation between materialized view partitions and patient data partitions allows us to fine tune indexes and reduce the amount of indexing required in our collection. We only need to activate indexing for:

1) The four metadata members of every document: *id*, *partitionKey*, *docType* and *docSubType*.
2) The items of materialized view partitions. To tell them apart from domain objects we have introduced the convention that the names of view objects start with “Query”, e.g. `QueryPatient, QueryOrder`

With these simple changes I’ve seen RU/s reductions in writes of 30% for a typical document. This is the final indexing policy configured for the single “nucorisCol” collection:

```
{
  "indexingMode": "consistent",
  "automatic": true,
  "includedPaths": [
    {
      "path": "/partitionKey/?",
      "indexes": [
        {
          "kind": "Range",
          "dataType": "String",
          "precision": -1
        }
      ]
    },
    {
      "path": "/docType/?",
      "indexes": [
        {
          "kind": "Range",
          "dataType": "String",
          "precision": -1
        }
      ]
    },
    {
      "path": "/docSubType/?",
      "indexes": [
        {
          "kind": "Range",
          "dataType": "String",
          "precision": -1
        }
      ]
    },
    {
      "path": "/docContents/QueryPatient/*",
      "indexes": [
        {
          "kind": "Range",
          "dataType": "Number",
          "precision": -1
        },
        {
          "kind": "Range",
          "dataType": "String",
          "precision": -1
        }
      ]
    },
    {
      "path": "/docContents/QueryOrder/*",
      "indexes": [
        {
          "kind": "Range",
          "dataType": "Number",
          "precision": -1
        },
        {
          "kind": "Range",
          "dataType": "String",
          "precision": -1
        }
      ]
    }

  ],
  "excludedPaths": [
    {
      "path": "/"
    }
  ]
}
```

6 The Persistence Layer Source Code

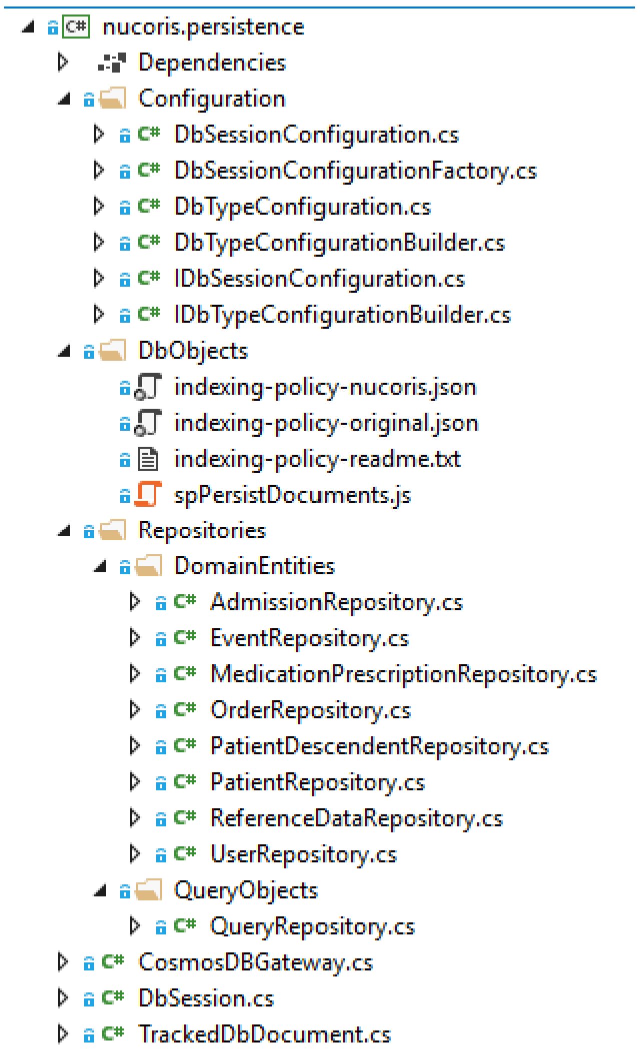
6.1 Project Overview

All **nucoris** code related to database access is located in the `nucoris.persistence` project, with the exception of repository interfaces which are defined in `nucoris.application.interfaces`.

I invite you to inspect the code on your own, but in this section I provide some guidance to make it easier. On the right you can see the structure of the project. Let's start with the folders:

- **Configuration**: contains classes encapsulating the description of how each entity is stored in the database (details in next section)
- **DbObjects**: contains the definition of the indexing policy and the script of the stored procedure we use to write the changes.
- **Repositories**: contains the implementation of the repositories for domain entities and materialized view items.

And three important classes explained later: `CosmosDBGateway`, `DbSession` and `TrackedDbDocument`



```
nucoris.persistence
  Dependencies
  Configuration
    DbSessionConfiguration.cs
    DbSessionConfigurationFactory.cs
    DbTypeConfiguration.cs
    DbTypeConfigurationBuilder.cs
    IDbSessionConfiguration.cs
    IDbTypeConfigurationBuilder.cs
  DbObjects
    indexing-policy-nucoris.json
    indexing-policy-original.json
    indexing-policy-readme.txt
    spPersistDocuments.js
  Repositories
    DomainEntities
      AdmissionRepository.cs
      EventRepository.cs
      MedicationPrescriptionRepository.cs
      OrderRepository.cs
      PatientDescendentRepository.cs
      PatientRepository.cs
      ReferenceDataRepository.cs
      UserRepository.cs
    QueryObjects
      QueryRepository.cs
  CosmosDBGateway.cs
  DbSession.cs
  TrackedDbDocument.cs
```

6.2 Configuration

As you've read in previous sections, each **nucoris** object is classified into one of four categories from the perspective of storage: *patient data*, *user data*, *reference data* and *materialized view data*.

Each category of data has different rules for assigning partition keys. These rules are centralized and encapsulated by a single class: `DbSessionConfigurationFactory`. This class creates the default implementation of `IDbSessionConfiguration`, which is injected at startup time into our unit of work, `DbSession`.

`DbSession` calls methods on the `IDbSessionConfiguration` instance to determine the collection and partition key that should be used to persist every object.

6.3 DbSession and TrackedDbDocument

`DbSession` is **nucoris**' unit of work. It's inspired by the "**unit of work controller**" pattern described by Martin Fowler in *Patterns of Enterprise Application Architecture* (pages 187-188 of its 2003 edition). In practice this means that **repositories do not access the database directly**: they *invoke the unit of work to load objects*, and *register edited objects in the unit of work* so they can later be written on commit.

`DbSession` has the following functions:

1) Gets collection/partition information from configuration.
2) Loads objects from the database on behalf of the repositories. It doesn't do it directly (it calls `CosmosDBGateway`), but being the middle-person allows it to track which objects have been read, and to move the collection/partition information away from repositories. Objects read from the database are embedded into instances of `TrackedDbDocument` so their state can be properly tracked.
3) Tracks added/modified objects, which are passed into DbSession by the repositories by calling `DbSession`'s `Register` method.
4) Provides a *Commit* method and specialized *Load* methods for specific items types that are later translated into low-level calls to `CosmosDBGateway`.

`DbSession` is expected to be used with request scope (*AsScoped()* in ASP.NET)

6.4 CosmosDBGateway

`CosmosDBGateway` encapsulates access to the Cosmos database. It uses Cosmos DB .NET SDK.

Writes are done by grouping pending changes by collection and partition key and passing each group of documents to the stored procedure `spPersistDocuments` we've created. Each document has a companion action state that indicates whether it's an insert, an update or a delete, so the procedure can invoke the appropriate Cosmos DB JavaScript method.

Object are serialized with the Newtonsoft's serialization setting `TypeNameHandling.Auto`, which embeds a $type property into the JSON document so that later it can be deserialized into the right subtype. For example:

```
"$type": "nucoris.domain.Patient, nucoris.domain"
```

`CosmosDBGateway` is expected to be used as singleton, according to Microsoft's recommendation.

7 Conclusions

After my experience in **nucoris**, **I recommend Cosmos DB when**:

1) Your domain has few relationships among entities, and the aggregate entities can be easily persisted as self-contained documents.
2) The domain has many entities with different properties (different shape/schema).
3) You have users around the world who may work with the same data at the same time, and you want to ensure a good performance for all of them, regardless of location. (Near real-time bidirectional replication is a problem that cannot be easily and reliably solved with a relational database.)
4) Your application must be highly available and scalable.
5) The queries you will run are few, simple and well-known.

An HIS does not fit well these criteria and therefore **a relational database such as Azure SQL would be more appropriate for an HIS than Cosmos DB**.
However, with the modeling approach I've described in this document it would be nevertheless feasible to use exclusively Cosmos DB as the data storage for an HIS, but I don't think it's the best choice.

nucoris is an exploratory prototype I've created to answer questions such as this one, and it is very satisfying to be able to reach a conclusion based on first-hand experience. **Cosmos DB is a great product that I've enjoyed working with**: do not hesitate to use it if your application matches the criteria listed above.