NosDB Whitepaper

MIGRATE .NET APPLICATIONS FROM SQL TO NOSQL

By

Alachisoft

December 2016

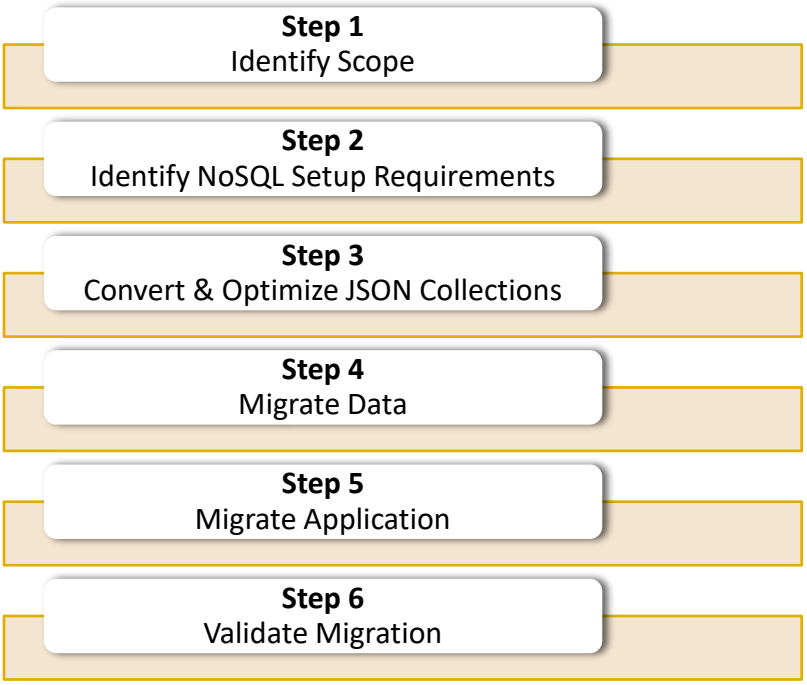
Table of Contents

Steps to Migrate Data from a Relational Database to NoSQL

Using a relational database in your .NET applications is not without some limitations, such as its inability to handle more load without replacing the existing hardware (scaling up) and its inability to update its rigid data model i.e. rows and columns. If you are facing these challenges, then you may have already decided to shift your database to NoSQL. If you're still not convinced, then you may want to read Why NoSQL?

Considering the case that you're out of options to scale your database layer and you have chosen to move your data to NoSQL, the daunting task of migration might be keeping you at bay from taking this leap.

While migration is a big task, if divided into steps, it can be approached in a very organized manner making your life easy. This whitepaper is here to help you organize your migration process in 6 logical steps.



If you follow these simple steps, they are bound to help you in your migration process. This paper considers that you already know the basics about the features of NosDB, a .NET NoSQL Document Database. If not then please head over to the website. Below I present you the details of the aforementioned process.

Step 1: Identify Scope

The first and the most important step is to understand your business model and the database schema. You will also need to completely understand how your application accesses this data and the data flow to-and-from your database. This helps identify two key pieces of information:

1. The part of your database schema that is most accessed (reads, writes or both)
2. Data that is grouped together i.e. always accessed together for reads and writes.

Having this information helps you identify key decisions in the remaining steps. Therefore, this is also the most crucial step.

Step 2: Identify NoSQL Setup Requirements

Once you're done understanding your business model and the data flow, you'll be in good shape to make a few rough decisions on how to deploy your NoSQL cluster, based on your business requirements. For example, noting down the current number of applications running, number of users and access rate are some important metrics. Noting which part of your data is highly sensitive i.e. you need to have a backup at all costs, also helps you decide the deployment strategy of the replica nodes.

Since a NoSQL database is a distributed database, you can leverage the distributed nature to your advantage, and deploy it per your business requirements. The biggest advantage you get, in addition to scalability, is distributing/deploying your NosDB cluster shards across different GEO locations. This allows you to deploy shards near your application's geographic location and avoid costly network trips. And it boosts app performance.

Just remember that all we're doing here is sketching out requirements. We will definitely revisit these settings after we identify, create and optimize our collections.

Step 3: Convert & Optimize JSON Collections

Now that you have the basic requirements of your deployment cluster, your ‘sketch’ will dictate your optimization strategies.

Convert Tables to Collections

First, simply convert the tables from the relational database into collections and convert your columns into attributes. This is the *normalized* data coming from a relational database.

De-normalize Data by Embedding Documents

Next, you need to *de-normalize* isolated tables, that is, tables which do not mean anything unless correlated with another table. In relational terms, isolated tables are *junction tables*. For example, `OrderDetails` in a Northwind database make more sense when mentioned with reference to an Order. Therefore, the right choice would be to embed the `OrderDetails` inside the `Order` documents.

Convert Relations

Now, what you have left are those collections containing documents which have their respective *junction tables* embedded within them. But what about `many-to-many`, `one-to-many` and other relations? This is where your knowledge about naturally-grouped-data and data-accessed-together comes in handy.

In the Migration of NorthWind Database Example, the `Customer` and `Order` tables are related but not always accessed together. So, it doesn't make sense to embed the `Customer` Object inside the `Order` Document. Furthermore, embedding the `Customer` Document will unnecessarily duplicate data, which we want to avoid as much as possible. Otherwise a single change in the `Customer` profile will require the application to make the change in all the `Order.Customer` documents. An unnecessary computation cost.

On the other hand, `Categories` are always required by the application whenever `Product` is fetched; therefore, this is a good candidate for embedding. And, do not forget that the beauty of JSON documents is that they can also support arrays, and arrays of JSON documents, to enrich your objects.

Hybrid Model - Best of Normalization and De-Normalization

Since a NoSQL schema is based on the application's data flow, if a collection has advantages of keeping it both embedded and non-embedded then adopt the hybrid model.

In the sample Northwind database, referenced on the previous page, this phenomenon can be seen in the `Category` and `Product` tables. The scenario is that whenever a `Product` is accessed, the application needs to know its `Category`. But the application also needs to find out `Products` by `Category`.

If the `Category` was kept in a separate table, a single product fetch would mean two database calls, one to fetch the `Product` and the other to fetch the respective `Category`, thus the extra network cost. If only the `Category` was embedded, then to find out all the categories you would have to run the following SQL statement:

```
SELECT DISTINCT product.Category.CategoryName FROM Products;
```

A simple SQL query but not a very efficient one is it? So the answer is to keep the `Category` document in a separate collection but embed only that portion which is necessarily required by the `Product`. This is called a **Hybrid Model**.

For example, in our scenario when the application fetches the `Product` document, it only requires knowing the `Category` Name and `Category` Description. Therefore, it is completely unnecessary to embed the `Category` Picture in every `Product` document. In fact, if duplicated, it will take a lot of unnecessary storage and increase the document size thus imposing a costly network trip.

This is a perfect use case of a Hybrid Model, thus the collections would be shaped as follows:

```
"Product" {
    "name": "string",
    "category": {
        "name": "string",
        "description": "string",
    }
}
```

and storing `Category` separately as well:

```
“Category" {
    "name": "string",
    "description": "string",
    "picture": byte[]
    }
}
```

Decide Your Distribution Strategy

The next thing to decide is the likely distribution strategy of your collections. The initial sketch of your setup requirements directly affects this.

You have three options for your distribution strategy:

- **Range Based Distribution:** This strategy allows you to define how data is distributed amongst the nodes according to the ranges specified for each shard. For example, if your NosDB cluster is GEO distributed with one shard in New York and another in London, then data generated and required by applications existing in New York should be in the same GEO location, thereby optimizing network costs. This strategy is used mostly in GEO distributed clusters but has other use cases too.
- **Hash Based Distribution:** Hashing allows you to distribute data across shards uniformly, thus uniformly distributing the load as well. This strategy is not the best choice for a GEO distributed cluster, but is great for NosDB clusters within a single data-center.
- **Single Sharded Collection (Disable Distribution):** This completely disables distribution on a collection. Use this option if your data set is small or you specifically want it to be in a single machine.

After deciding your distribution strategy, you might want to revisit your collection optimization and your deployment strategy to see if they can be further optimized. A couple iterations are usually enough to reach a decision.

Step 4: Migrate Data

Finally, after some heavy brainstorming, here comes the relatively easy part i.e. migrating data from the relational database to the NoSQL database.

First create your .NET objects representing your collections and JSON documents. Yes! no ORM is needed to insert data since the .NET API automatically converts your .NET objects to JSON Documents. (Just a note, however, you could also opt to use the ADO.NET integration shipped along with NosDB).

Next, access your relational database and populate these .NET Objects, and insert them into the NoSQL database. You can also use CLR Triggers and CLR UDFs provided by NosDB to assist your migration.

Once you have your data migrated, now is the time to migrate your application to adopt the data in terms of collections and documents. Without NosDB you do not get the option of using ADO.NET or CLR Triggers & UDFs but you could still use the API.

Step 5: Migrate Application

There are multiple ways to migrate your working .NET application to NosDB. NosDB supports SQL operations like SELECT, INSERT, UPDATE and DELETE. Using SQL operations greatly reduces the learning curve to migrate your application i.e you can use the syntax you are used to. You can even manage the database cluster in NosDB using SQL.

NosDB supports SQL with all the multiple ways the database can be accessed, namely:

- .NET API
  - ADO.NET
  - LINQ
- Java API
- REST API

You can also use the server side API to improve your application performance, leveraging the power of distribution by using frameworks such as MapReduce. If you're not using NosDB you only have the option of calling the API directly. SQL and ADO.NET are only provided by NosDB.

Step 6: Validate Migration

After migration, validation is the last step in the whole process: verify all your tests, validate migrated data and your application. This step is completely up to you and your business processes. Bench the newly incorporated NoSQL database. Check the limits of the current cluster configuration (although you can scale out whenever you want to) and equip yourself with the right tools like NosDB Management Studio to manage and monitor the whole cluster from a single location.

That's it! If you follow these steps you can organize your migration from a relational database to a NoSQL database in a logical 6 step process.

IMPORTANT LINKS

- What is a Document Database?
- Why NoSQL?
- Migrating a Northwind Database to NoSQL
- Triggers in a NoSQL Database
- Getting Started with NosDB
- Download NosDB