Migration from PostgreSQL to MongoDB on AWS

Introduction

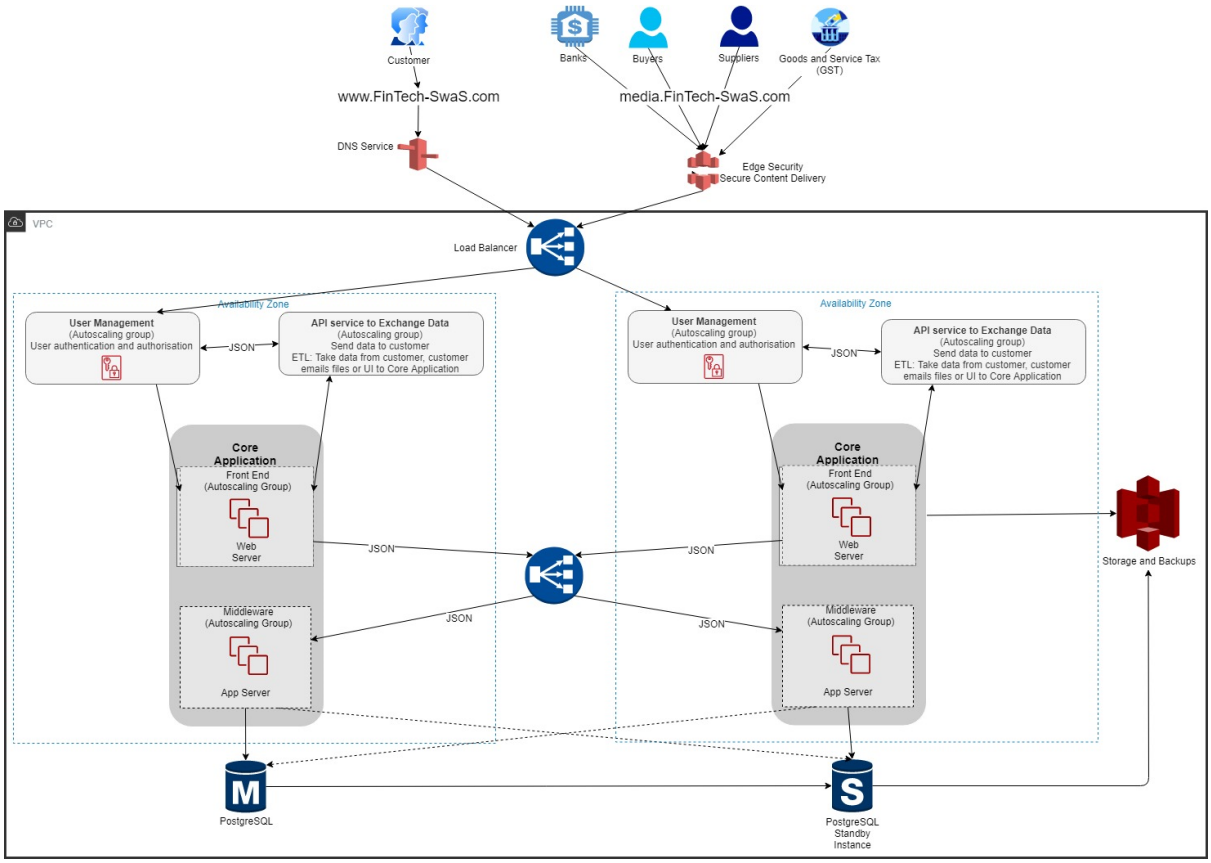
I have a friend who is the Chief Technical Officer (CTO) in a Financial Technology (FinTech) start-up that provides a proprietary Software with a Service (SwaS). When he defined their product, a relational database was a natural fit as the backend for their application, so he chose PostgreSQL.

However, as their strategy, goals and product evolved, my friend realised that their application needed a plastic data model, which a relational database can't provide, so they must migrate to a NoSQL data store.

I have been supporting my friend on the architecture and planning for this migration. This post describes the process I followed in choosing the right NoSQL database as per this FinTech requirements, proposes an architecture for this SwaS based on MongoDB (the target database), and explains how to migrate.

Current Application Architecture

You can see below the high-level architecture diagram of the SwaS that my friend's company provides. It is a web application with the front end, middleware, and a PostgreSQL database as backend. It runs on AWS Public Cloud, and it is designed for high availability and resiliency.



Requirements

The requirements to improve this application are:

1. The data model needs to be flexible,
2. Minimise the development team's effort to migrate the application,
3. Perform the migration as quickly as possible to avoid getting on the way of the fast pace of this start-up, and
4. Minimise the cost of moving the database.

Choosing the Target NoSQL Database

A NoSQL (initially referring to "non SQL" or "non relational") database provides a mechanism for storage and retrieval of data which is an alternative to the tabular relations used in relational databases.

These databases have existed for long. However, they have become trendy in the latest years, where different database technologies have been developed in response to the demands of building applications as Web 2.0, analytics, IoT, etc.

Neither of these databases replaces RDBMS[1], nor are they ACID databases[2]. Thus, if you have a transactional workload where normalisation and consistency are the primary requirements, a NoSQL database will not be the right choice for you.

The NoSQL data stores are a heterogeneous group of databases. Generally speaking, they are classified in the next groups: Column, Document, Key-value and Graph[3]. Even though on some occasions, the classification is more detailed[4].

However, the distinction between NoSQL data stores types is not so clear. One database may be based on a key-value architecture but implement some features of a document or a graph NoSQL store.

There are many NoSQL data stores[5]. After some research, I deep dived into the next ones because they are used in several well-known implementations, they can be easily integrated into the AWS cloud, and they seem a priori a good fit for this company's requirements.

- MongoDB – document-oriented NoSQL database,
- Apache Cassandra – column-store NoSQL database, and
- AWS DynamoDB –key-value store NoSQL database.

Next, there is a detailed explanation of the different features of these databases. If you need an overview of the features to choose one data store above the others, jump to the comparative summary of the three databases.

[1] RDBMS stands for Relational Database Management System
[2] ACID (Atomicity, Consistency, Isolation, Durability) is a set of properties of database transactions intended to guarantee validity even in the event of errors, power failures, etc.
[3] Wikipedia – NoSQL.
[4] Christof Strauch. "NoSQL Databases". Hochschule der Medien University, Stuttgart
[5] For example, the Forrester Wave™: Big Data NoSQL, Q1 2019 focuses on Document-oriented NoSQL databases.

MongoDB

MongoDB was started in 2007 by 10gen, which created the product based on the word "humongous". In 2009, it was released, and 10gen later changed their company name to MongoDB, Inc.

It is probably the most popular NoSQL database. It provides scalability and caching for real-time analytics. Additionally, if there is no clear schema definition, MongoDB can be the right choice. However, it is not built for transactional data (accounting systems, etc.).

MongoDB is frequently used for mobile apps, content management, real-time analytics, and applications involving the Internet of Things.

It stores data in documents, like XML or JSON. These documents can have varied structures. Since it is schema-free, we can create documents without having to create the structure for the document first.

JSON format allows for data hierarchy and high flexibility of the data model, and it is the key to MongoDB's support to a rich and expressive object model.

MongoDB's model is based on objects, which can have properties, and they can be nested in one another (for multiple levels).

Query Language

MongoDB has its language, MongoDB query language, to access data, which is quite different from SQL. On the positive side, there are free tools which help to translate code from SQL to MongoDB[6].

Indexes

Indexes are an advanced and powerful feature in MongoDB. If an index is missing, every document within a collection must be searched to select the documents that were requested in the query, and the read times will be slower.

Additionally, indexes perform well on any property of an object stored in MongoDB, even on nested objects.

Replication

On a separate note, MongoDB has built-in replication with auto-elections. This feature allows to set up a secondary database that can be auto elected if the primary database becomes unavailable.

MongoDB has replica sets where one member is the primary and all others have a secondary role. The reads and writes are committed to the primary replica first and then replicated to the secondary replicas.

MongoDB supports a "single master" mode, which means there are a master node and several slave nodes. In case the master goes down, one of the slaves is elected as master. This process happens automatically, but it takes time, usually 10-40 seconds. During this time the replica set is down, and it doesn't take writes.

[6] For example, the SQL-to-MongoDB tool provided by site24x7.com.

Write Scalability

MongoDB, with its "single master" model, can take writes only on the primary node. The secondary servers can only be used for reads. That's to say, if there are three node replica set, only the master is taking writes, and the other two nodes are only used for reads.

Native Aggregation for ETL[7]

MongoDB has a built-in aggregation framework to run an ETL pipeline to transform the data stored in the database. This framework is useful for small to medium jobs. However, as data processing needs become more complicated, the aggregation framework becomes challenging to debug.

Support & Documentation

MongoDB has enterprise-grade support that provides 24 x 7 support along with the option for extended lifecycle support.

Extended lifecycle support allows us to continue using older versions.

Getting support from MongoDB gives you unlimited access to security fixes and updates.

MongoDB, Inc. maintains the MongoDB documentation. There is information about the MongoDB Server, Atlas (Database as a Service for AWS, Azure and Google), cloud manager for hosted MongoDB, and Ops Manager.

Supported languages

Actionscript, C, C#, C++, Clojure, ColdFusion, D, Dart, Delphi, Erlang, Go, Groovy, Haskell, Java, JavaScript, Lisp, Lua, MatLab, Perl, PHP, PowerShell, Prolog, Python, R, Ruby, Scala, Smalltalk.

Apache Cassandra

Avinash Lakshman and Prashant Malik developed Cassandra at Facebook for the Facebook inbox search feature. Facebook released Cassandra in July 2008 as an open-source project. The original developers got the name for the project from Cassandra, a Trojan mythological prophet. The Apache Software Foundation is currently behind the database.

Cassandra stores data in columns. Its table structure is a fairly traditional one with rows and columns. Data is more structured than MongoDB, and each column has a specific type which can be specified during creation. The columns' names and formats can be changed.

Column families are similar to tables in RDBMS, and each has a unique key. Unlike a traditional RDBMS, all rows in a table are not forced to have the same columns, and columns can be added on the fly.

Cassandra is optimised for high speed.

One of Cassandra's greatest strengths is its ability to scale while still being reliable. It is possible to deploy Cassandra across multiple servers built-in without much extra work. Part of this is due to how Cassandra handles replication. This feature makes Cassandra a good option for those data stores which significantly grow in a short period.

Cassandra is not meant for transactional data (i.e., accounting systems).

[7] ETL stands for Extract, Transform, Load process

Query Language

Data is accessed using the Cassandra Query Language (CQL), which is similar to SQL in syntax. This characteristic of CQL does not make Cassandra a relational database. There are different methods to store and retrieve data.

CQL is not entirely ANSI SQL. It has several limitations, as no join support, no OR clauses, etc.

Indexes

Cassandra has only cursory support for secondary indexes. Secondary indexes are also limited to single columns and equality comparisons. You can only query using the primary key.

Replication

Cassandra does replication out-of-the-box. The number of nodes is configured during the setup process. Cassandra automatically copies data to all nodes, and it takes care of the rest of the process.

Cassandra allows for multiple masters where losing a single node still lets you write to the cluster. This feature can allow for better fault tolerance without the 10 to 40-second downtime required with MongoDB.

Write Scalability

Cassandra can take writes on any server.

Native Aggregation for ETL

Cassandra does not have a built-in aggregation framework. External tools like Hadoop, Spark are used for ETL.

Support & Documentation

Support for Cassandra comes from third-party companies like Datastax, URImagination, Impetus, and more. A complete list of Cassandra DB third-party support providers can be found at here.

The Apache Software Foundation maintains the Cassandra documentation. There is information on how to get started with Cassandra, the Cassandra Query Language, Tools, FAQS, and more.

Additionally, Datastax also maintains documentation at their site.

AWS DynamoDB

In September 2013, Amazon made available a local development version of DynamoDB so developers can test DynamoDB-backed applications locally. It is a part of Amazon Web Services.

DynamoDB uses tables, items and attributes as core components. A table is a collection of items, and each item is a collection of attributes. It does not support embedded data structures, like with MongoDB.

DynamoDB uses primary keys to uniquely identify each item in a table and secondary indexes to provide more querying flexibility.

To write or retrieve data, we must use DynamoDB own language, which is no SQL-like.

Indexes

In DynamoDB, you can create and use a secondary index for similar purposes as in RDBMS. A secondary index has key attributes, defined when the index is created.

DynamoDB does not have a query optimiser, so a secondary index is only used when querying or scanning.

Replication

AWS provides a cross-region replication solution based on an open-source command-line tool. Detailed information is on GitHub.

The DynamoDB cross-region replication solution uses the Amazon DynamoDB Cross-Region Replication Library. This library uses DynamoDB Streams to sync DynamoDB tables across multiple regions in near real-time. When a DynamoDB table is written in one region, those changes are automatically propagated by the Cross-Region Replication Library to all tables in other regions.

Support

Amazon provides support via the Community Support Forum, ServerFault, and StackOverflow. Enterprise support is also available.

Supported languages

Java, JavaScript, Swift, Node.js, .NET, PHP, Python.

Comparative Summary of the Three Databases

	MongoDB	Cassandra	AWS DynamoDB
Data Model	Rich data model - can easily represent any object structure	Fairly traditional table structure	Not so traditional table structure
Data Storage	Documents	Flexible wide-column	Table structure
Secondary indexes	First-class construct	Cursory support. You better query on primary keys	No optimiser. Secondary indexes only used when querying or scanning
Scalability	Requires some work	Easy and reliable	
Replication	Built-in replication. A fair amount of work to setup	Built-in replication. Minimum set up	Cross-region replication solution based on an open-source command-line tool
Write Scalability	Writes only on the primary master node	Writes on any server	
Native Aggregation	Built-in aggregation framework to run an ETL pipeline	Not integrated aggregation framework. External tools like Hadoop, Spark are used	
Syntax	MongoDB language is very different from SQL – Free tools to translate code.	CQL is very similar to SQL	AWS DynamoDB language is very different from SQL
Support	Enterprise-grade support that provides 24 x 7	Support for Cassandra comes from third-party	Enterprise support

	MongoDB	Cassandra	AWS DynamoDB
	support along with the option for an extended lifecycle support	companies like Datastax, URImagination, Impetus, and more	
Supported Languages	Actionscript, C, C#, C++, Clojure, ColdFusion, D, Dart, Delphi, Erlang, Go, Groovy, Haskell, Java, JavaScript, Lisp, Lua, MatLab, Perl, PHP, PowerShell, Prolog, Python, R, Ruby, Scala, Smalltalk		Java, JavaScript, Swift, Node.js, .NET, PHP, Python
Portable to different platforms	Yes	Yes	Locked up to AWS

Conclusion

The reason to change the backend of this application is to get a flexible and rich data model. To that end, MongoDB is the better fit from my friend's needs.

Moreover, MongoDB is based on JSON. This feature will reduce language conversion in my friend's scenario, where data is exchanged among different ecosystem's elements in JSON format.

Furthermore, the native aggregation framework to load data will be instrumental in this environment.

Because of the secondary indexes features MongoDB provides, I expect the queries' response times are short.

Since it is so popular, it is supported by many tools and applications, and it allows to use many programming languages. Hence it will be easier to integrate into future scenarios.

Additionally, MongoDB can be ported to another platform if needed.

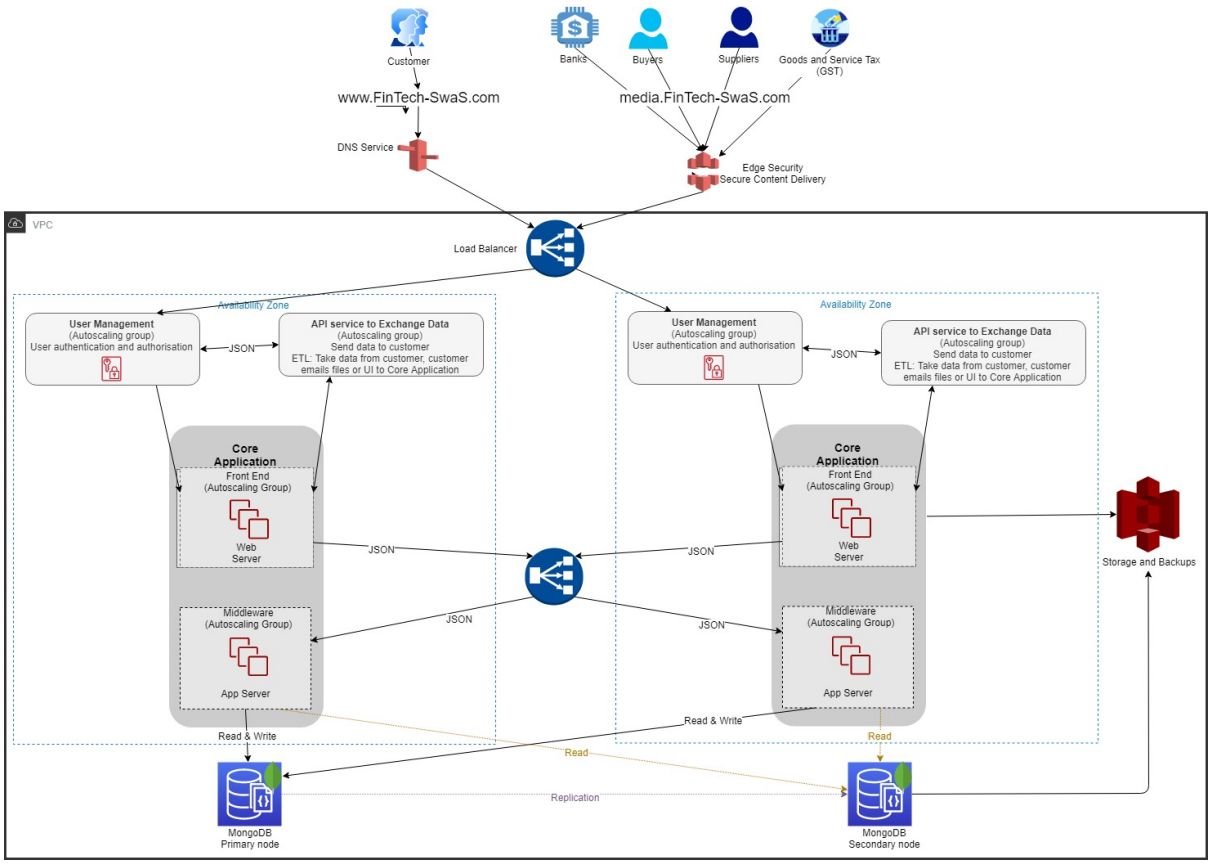
Finally, since MongoDB is schema-free, we can create documents without having to create the structure for the document first. This feature will be very convenient for the migration, and to quickly upload new data and test new functionality in the feverish pace of development a start-up has.

Architecting and Planning a Database Migration

See my post Planning a Database Migration to understand what a Database Migration implies, how to architect the details specific to your scenario, and how to plan it.

Future Application Architecture

After the migration, the architecture of the Software as a Service my friends provide will look as in the diagram below. It will continue running on AWS.



How to Migrate the Application and Database to MongoDB

There are several decisions to plan and architect a migration which depends on your particular business needs, priorities, ecosystem, functionality, etc. In this section, I am going to discuss how to make a bulk data movement from PostgreSQL MongoDB migration. I have recommended this approach to this FinTech company to migrate its repository.

I have not developed the tool to do this migration. However, it is relatively simple to create the commands from a PostgreSQL inventory. If we invest a bit more work, we can create a framework for the migration tool, which includes logs and restarts options in case of failure.

A bulk data movement means that I extract all information in the tables from PostgreSQL at once, and load it in MongoDB. There is the possibility to export the data and insert it one row at a time. The inconvenience of moving a row at a time is:

1. You may have poor performance, depending on your network configuration, if you are working on the cloud and how your setup is, the volume of data, etc., and
2. 2. You may lock your data during the migration unless you prevent all users from accessing your database during all the data movement. Depending on your scenario (amount of data, business requirements, time for the migration), this could not be feasible.

Typically a bulk copy is faster. However, it would be best if you prevent users from connecting to the database when you extract the data from PostgreSQL. It should be shorter than the other option. Still it would help if you decided the strategy to allow users to connect and catch up with work, as I explained in my post Planning a Database Migration.

Test Database in PostgreSQL

I am going to post some code I have prepared to support this FinTech migration to MongoDB. I have made my tests on the next virtual machines by Bitnami:

- bitnami-postgresql-12.0.0-1-r01-linux-debian-9-x86_64-nami.ova
- bitnami-mongodb-4.2.1-0-linux-debian-9-x86_64.ova

Once I launched the PostgreSQL database, I connected to it following these instructions and these ones.

Then I created a test environment.

```
bitnami@debian:~$ psql -d postgres -U postgres -W
Password:
psql (12.0)
Type "help" for help.

postgres=# CREATE DATABASE test_migration;
CREATE DATABASE
postgres=# CREATE TABLE students(id INTEGER, s_name VARCHAR(20));
CREATE TABLE
postgres=# CREATE TABLE classes(id INTEGER, subject VARCHAR(20));
CREATE TABLE
postgres=#
postgres=#
postgres=# INSERT INTO students(id, s_name) VALUES (1, 'Tanit'), (2,
'Ishtar');
INSERT 0 2
postgres=# INSERT INTO classes(id, subject) VALUES (1, 'Math'), (2,
'Science'), (3, 'Biology');
INSERT 0 3
postgres=#
```

Inventory PostgreSQL Database

In order to plan for the migration, it is necessary to calculate space and inventory the PostgreSQL database. We can use psql for these tasks. With a little help of a text editor, Excel or a shell script, we can format the output to better read it, and even prepare the commands we will run to extract the data from PostgreSQL.

```
bitnami@debian:~$ psql -d postgres -U postgres -W
Password:
psql (12.0)
Type "help" for help.

postgres=# \l
                                  List of databases
      Name       |  Owner   | Encoding | Collate | Ctype |   Access
privileges
-----------------+----------+----------+---------+-------+-----------
------------
 postgres        | postgres | UTF8     | C       | C     |
```

```
 template0      | postgres | UTF8     | C       | C     |
=c/postgres           +
                |          |          |         |       |
postgres=CTc/postgres
 template1      | postgres | UTF8     | C       | C     |
=c/postgres           +
                |          |          |         |       |
postgres=CTc/postgres
 test_migration | postgres | UTF8     | C       | C     |
(4 rows)

postgres=# \l+

List of databases
      Name       |  Owner   | Encoding | Collate | Ctype |   Access
privileges   |  Size   | Tablespace |
        Description
---------------+----------+----------+---------+-------+-----------
-----------+---------+------------+--------
-----------------------------------
 postgres       | postgres | UTF8     | C       | C     |
| 7953 kB | pg_default | default
 administrative connection database
 template0      | postgres | UTF8     | C       | C     |
=c/postgres           +| 7929 kB | pg_default | unmodif
iable empty database
                |          |          |         |       |
postgres=CTc/postgres |         |            |
 template1      | postgres | UTF8     | C       | C     |
=c/postgres           +| 7785 kB | pg_default | default
 template for new databases
                |          |          |         |       |
postgres=CTc/postgres |         |            |
 test_migration | postgres | UTF8     | C       | C     |
| 7785 kB | pg_default |
(4 rows)

postgres=# \dn
  List of schemas
  Name  |  Owner
--------+----------
 public | postgres
(1 row)

postgres=# \dn+
                          List of schemas
  Name  |  Owner   |  Access privileges   |      Description
--------+----------+----------------------+------------------------
 public | postgres | postgres=UC/postgres+| standard public schema
        |          | =UC/postgres         |
(1 row)
```

```
postgres=# \d
          List of relations
 Schema |   Name   | Type  |  Owner
--------+----------+-------+----------
 public | classes  | table | postgres
 public | students | table | postgres
(2 rows)

postgres=# \d+
                        List of relations
 Schema |   Name   | Type  |  Owner   |    Size    | Description
--------+----------+-------+----------+------------+-------------
 public | classes  | table | postgres | 8192 bytes |
 public | students | table | postgres | 8192 bytes |
(2 rows)

postgres=# \d students
                    Table "public.students"
 Column |         Type          | Collation | Nullable | Default
--------+-----------------------+-----------+----------+---------
 id     | integer               |           |          |
 s_name | character varying(20) |           |          |

postgres=# \d+ students
                                             Table "public.students"
 Column |         Type          | Collation | Nullable | Default |
Storage  | Stats target | Description
--------+-----------------------+-----------+----------+---------+--
--------+--------------+-------------
 id     | integer               |           |          |         |
plain    |              |
 s_name | character varying(20) |           |          |         |
extended |              |
Access method: heap

postgres=#
```

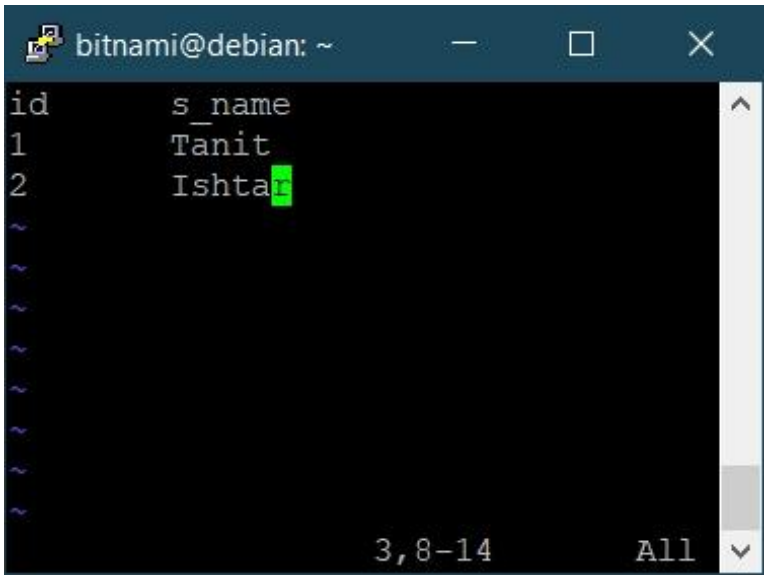
Amendment November 8th, 2019: I have published a post with an inventory for a PostgreSQL cluster which will fit the requirements for migration.

Migrate Data and Objects

To migrate data and objects in my test, I first extract the data in PostgreSQL with COPY command.

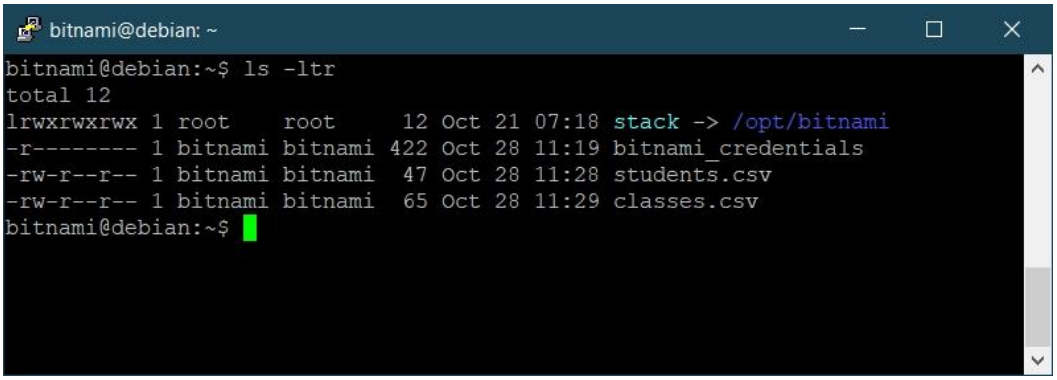
```
bitnami@debian:~$ psql -d postgres -U postgres -W
Password:
psql (12.0)
Type "help" for help.

postgres=# COPY (
postgres(#     SELECT id, s_name
```

```
postgres(#     FROM
postgres(#         public.students
postgres(# ) TO '/tmp/students.csv' WITH (FORMAT CSV, HEADER TRUE,
DELIMITER E'\t');
COPY 2
postgres=#
postgres=# COPY (
postgres(#     SELECT id, subject
postgres(#     FROM
postgres(#         public.classes
postgres(# ) TO '/tmp/classes.csv' WITH (FORMAT CSV, HEADER TRUE,
DELIMITER E'\t');
COPY 3
postgres=# exit
bitnami@debian:~$ ls -ltr /tmp/
total 36
-rw-r--r-- 1 root     root  170 Oct 28 09:13 nami_1572253989.log
-rw-r--r-- 1 root     root  162 Oct 28 09:13 nami_1572253990.log
-rw-r--r-- 1 root     root 1899 Oct 28 09:13 nami_1572253992.log
-rw-r--r-- 1 root     root   84 Oct 28 09:13 nami_1572254001.log
-rw-r--r-- 1 root     root    0 Oct 28 09:13 nami_1572254003.log
drwx------  3 root         root  4096 Oct 28 10:12 systemd-private-
b7c4e1229b2044ff977f15f43a582add-haveged.service-JHSaDR
drwx------  3 root         root  4096 Oct 28 10:12 systemd-private-
b7c4e1229b2044ff977f15f43a582add-systemd-timesyncd.service-jVMTLP
drwx------ 2 root     root 4096 Oct 28 10:12 vmware-root
-rw-r--r-- 1 postgres root   27 Oct 28 10:42 students.csv
-rw-r--r-- 1 postgres root   38 Oct 28 10:42 classes.csv
bitnami@debian:~$ vi /tmp/students.csv
```

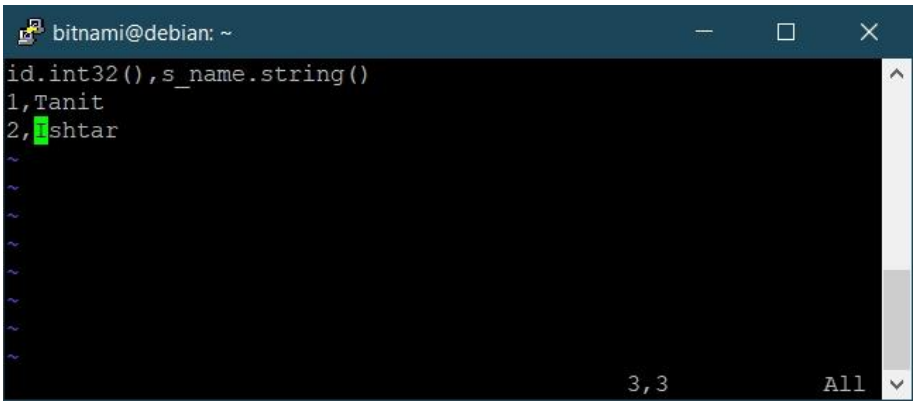


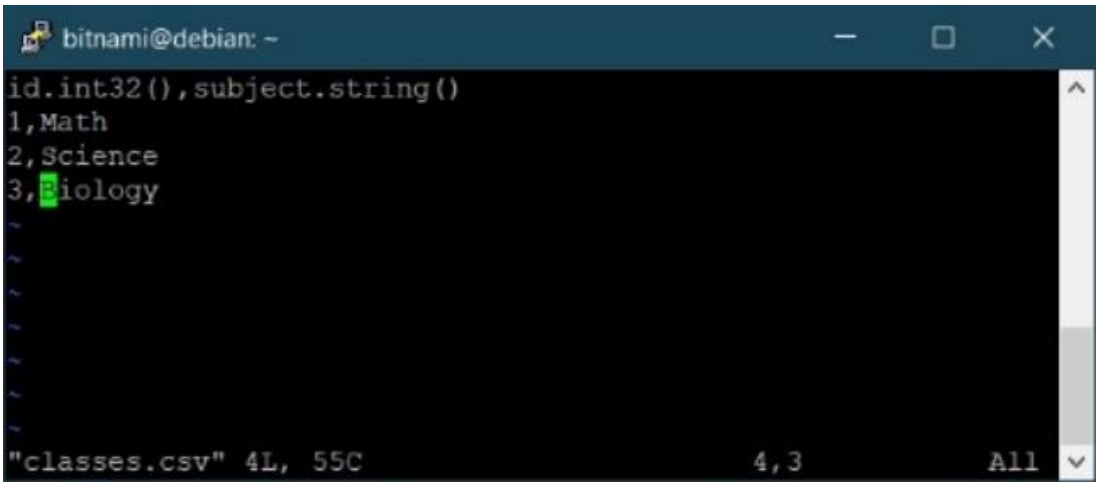
Now I launched my Bitanmi's MongoDB virtual machine, and I connected to it following these instructions and these ones.

Now I can SFTP the files from the PostgreSQL server to the MongoDB one.



I edit the data files to include the fields data types. I can add this step in the automatic generation of data files and commands to import, as mention in the previous section.





I import data into MongoDB.

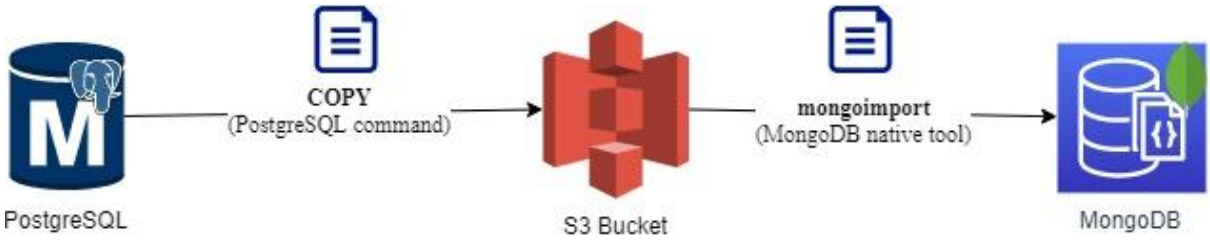
```
bitnami@debian:~$ mongoimport --username=root --password=b7roG4Bo2N1Q --authenticationDatabase=admin --db=newdatabase --collection=students --type=csv --file=students.csv --headerline --columnsHaveTypes --ignoreBlanks

2019-10-28T12:10:38.397+0000    connected to: mongodb://localhost/

2019-10-28T12:10:38.412+0000    2 document(s) imported successfully. 0 document(s) failed to import.

bitnami@debian:~$ mongoimport --username=root --password=b7roG4Bo2N1Q --authenticationDatabase= admin --
```

```
db=newdatabase --collection=classes --type=csv --file=classes.csv --
headerline --columnsHaveTypes –ignoreBlanks

2019-10-28T12:10:50.936+0000    error validating settings: only one
positional argument is allowed

bitnami@debian:~$
```

If data types are not defined in the data files, I can add next modifiers to the mongoimport commands:

```
mongoimport --username=root --password=b7roG4Bo2N1Q --
authenticationDatabase= admin --db=newdatabase --collection=students
--type=csv --fields id.int32,s_name.string --file=students.csv --
headerline --columnsHaveTypes --ignoreBlanks

mongoimport --username=root --password=b7roG4Bo2N1Q --
authenticationDatabase= admin --db=newdatabase --collection=classes
--type=csv --fields id.int32,s_subject.string --file=classes.csv --
headerline --columnsHaveTypes –ignoreBlanks
```

By the time I wrote this post, the recommendation was to use ESB (Elastic Blob Storage) storage for MongoDB on AWS. Since we are going to migrate a significant volume of data, and I don't want to overload the ESB for my MongoDB, we are going to use an S3 bucket as a staging area to extract and load the data. We will create an S3 bucket dedicated to the migration, we will connect MongoDB and PostgreSQL to it, and we will use the COPY command and mongoimport to move the data.



In our scenario, we are going to migrate data within a VPC and a region. Data will be protected as per this FinTech requirements. Thus, we won't encrypt the files in transit.

On a separate note, we have not tested the data movement speed by the time of writing this post. Once we have this information, we will decide if we need to compress the data files.

Translating Code

As part of the migration effort, we must migrate the queries running on PostgreSQL.

In some occasions, there is software available to automatically translate queries from one database vendor to another one on the flight. This software is installed in a server between the applications (middleware, APIs, etc.) and the target database, and when any query runs on the clients with the source SQL language, it is transformed on the flight to the new datastore language. This solution speeds up the migration a lot, even though it requires to set up a new server, the queries may not be optimum, and it is necessary to pay an additional license.

I couldn't find such a software for a PostgreSQL-to-MongoDB migration, so in this case, the only option is to change all code run on the datastore from SQL to MongoDB language. The migration will be more extended, but we won't need to pay for another tool, and the code we will eventually run on MongoDB will be better.

To support us on the task of translating code, we can use free automatic tools, for example, the convertor provided by www.site24x7.com.

Connecting Applications to MongoDB

The Spring Data project provides with a consistent programming model for data accessed, and it is the preferred method for connecting applications to MongoDB. The Spring Data MongoDB reference documentation is here, and there is also an example of how to do it.

This FinTech company uses JPA (Java Persistent API) to connect their product with PostgreSQL. Many JPA implementations support connections to NoSQL datastores, as Hibernate OGM and EclipseLink. Here you have an example on how to use Hibernate OMG on MongoDB.

Some sources familiar with JPA and MongoDB believe that JPA does not support many MongoDB features. They also think that just switching the store behind object model is not the solution. See this post.

These same sources recommend moving to the dedicated Spring Data MongoDB project, which counts with Spring Data JPA. Its documentation is here. If you want to read a justification to move to Spring Data JPA, see this article.