Migrating data from MySQL to MongoDB

Introduction

Relational databases such as the MySQL databases have played an important role in data storage, data query and data management (Yoo et al. 2018, p. 243). However, with the rise of big data and Web 2.0, the limitations of relational databases have become more apparent. Query performance begin to decline with huge amounts of data, especially when highly normalised relational databases are difficult to scale horizontally (Tang & Fan 2016, p. 105).

Due to these drawbacks, NoSQL databases were developed. Compared to relational databases, NoSQL databases, do not store their data in tabular relations, making it more flexible and highly scalable (MongoDB 2019). NoSQL databases do not have have to follow a strict schema. They can handle large volumes of structured, semi-structured, and unstructured data.

With such large number NoSQL databases becoming available, it is important to understand the features they provide and what differentiates them. NoSQL technology has been rapidly developed with different implementation mechanisms, storage characteristics and optimisation methods, bringing more challenges to selecting the right NoSQL database.

ACID vs BASE Consistency Models

Relational databases are based on a set of principles to minimise performance. It is known as the ACID consistency model (Abramova & Bernardino 2013):

- **A**tomic - All operations in the transaction succeed or every operation is rolled back
- **C**onsistent - On the completion of a transaction, the database is structurally sound.
- **I**solated - Transactions do not conflict with another.

- **D**urable - The results of applying a transaction are permanent and cannot be undone.

This consistency model is to ensure the database is robust but ACID is much harder when there the amount of data is much larger. This is why NoSQL databases follow the BASE consistency model:

- **B**asic **A**vailability - The database is distributed, even when there is a failure the system continues to work.
- **S**oft-state - consistency is not guaranteed, different replicas don't have to be consistent.
- **E**ventual consistency - guarantees that even when data is not consistent, it will be eventually.

It is impossible to state which consistency model is superior because of their different uses. The BASE model is more flexible that ACID but that comes at the expense of consistency. If consistency is important, an ACID database would be better suited for the job rather than BASE (Sasaki 2018). An example of when ACID would be chosen would be banking, can you imagine what would happen if a person's data on two servers were different?

Types of NoSQL databases

According to MongoDB (2019), there are four main types of NoSQL databases are:

- Key-Value: Every item in the database is stored as a key together with its rank. Some examples of this type are Riak, Voldemort, and Redis.
- Wide-column stores: Store data together as columns instead of rows and are optimised for queries over large datasets. Some examples of this type are Cassandra and HBase.
- Document: Pair each key with a complex data structure known as a document. Documents can contain many different key-value pairs, or key-array pairs, or even nested documents (DB-Engines 2019). Some examples of this type are Couchbase and MongoDB.
- Graph-Oriented: Used to support information about networks. Some examples of this type are Neo4J and HyperGraphDB.

MySQL

MySQL is an open-source relational database management system (RDBMS), which stores data in tables and maintains a relationship between the data. It uses a powerful querying language, SQL (Structured Query Language), to communicate with databases, using commands such as 'SELECT', 'UPDATE', 'INSERT', 'DELETE' and many more. Related information that is stored in different tables, and using the 'JOIN' operation can be combined using values in common to each. MySQL is suitable for a highly transactional system and application with traditional database requirements such as foreign key constraints (Babu & Surendran 2017). Data in MySQL are organised and conforms to a certain format, making it one of the most popular structured databases today (DB-Engines 2019).

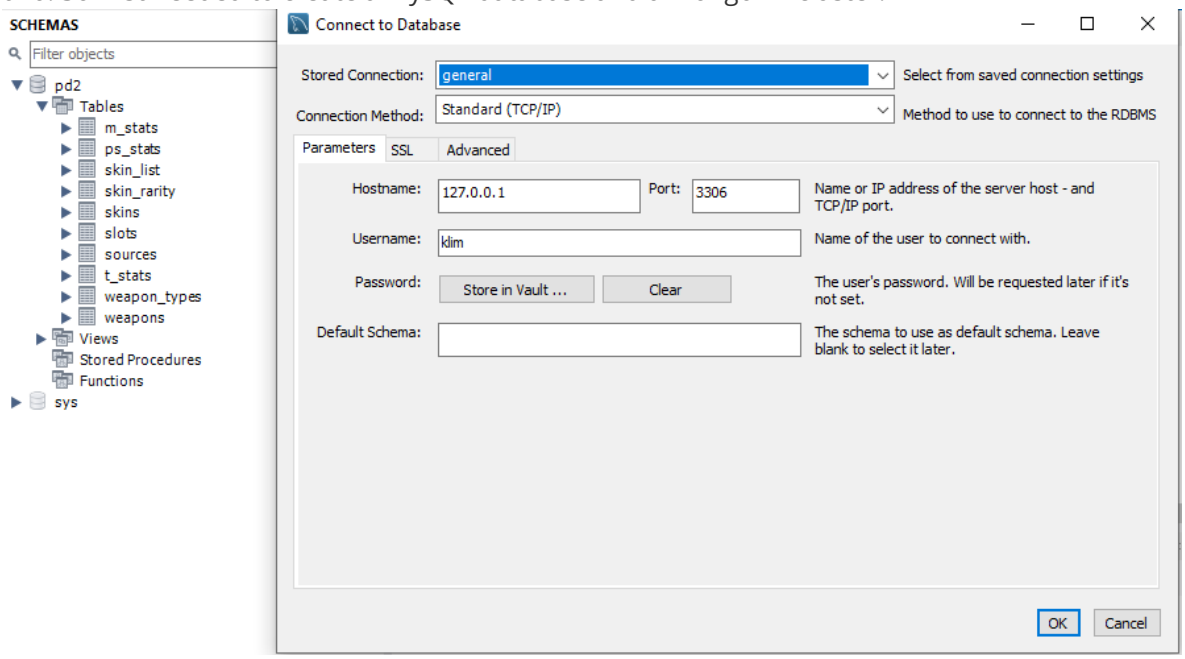
MongoDB

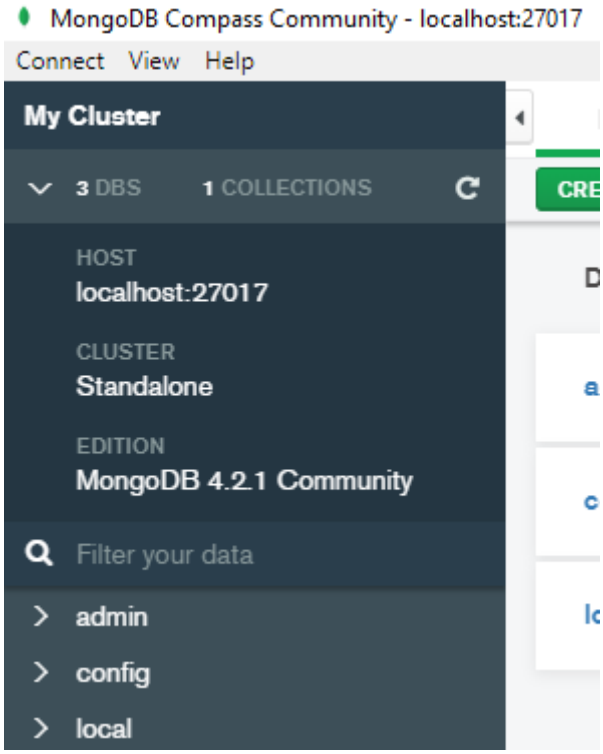
MongoDB is the most popular open-source document-oriented database at the moment [DB-Engines 2019]. MongoDB can store related data together, which is accessed using the MongoDB querying language. Instead of tables, MongoDB stores its data in a collection. Documents are essentially tuples that you can find in a relational database table (MongoDB 2019). Adding new columns to a MySQL database can lock up the entire database and cause performance issues. The schema-less MongoDB, on the other hand, can add new fields without effecting existing rows. To do this in a relational database such as MySQL would require the table to restructure. MongoDB is suitable as a RDBMS replacement for web applications, semi-structured content

management, real-time analytics, high-speed logging, caching and high scalability (Babu & Surendran 2017).

Methodology

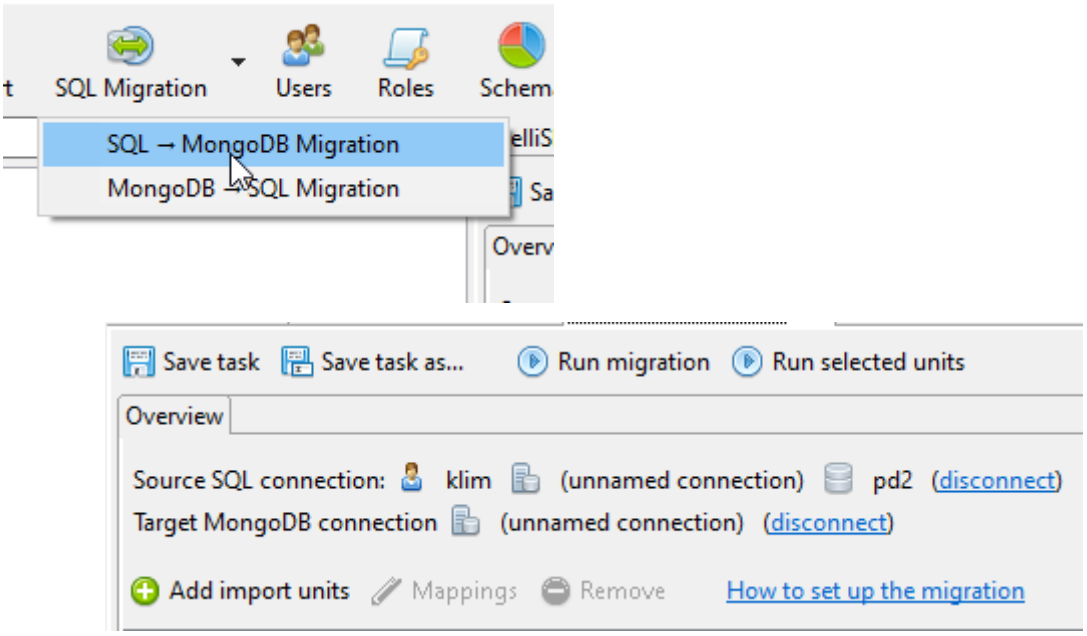
To migrate the database from MySQL to MongoDB, there needed to be a connection between the two. So first needed to create a MySQL database and a MongoDB cluster.



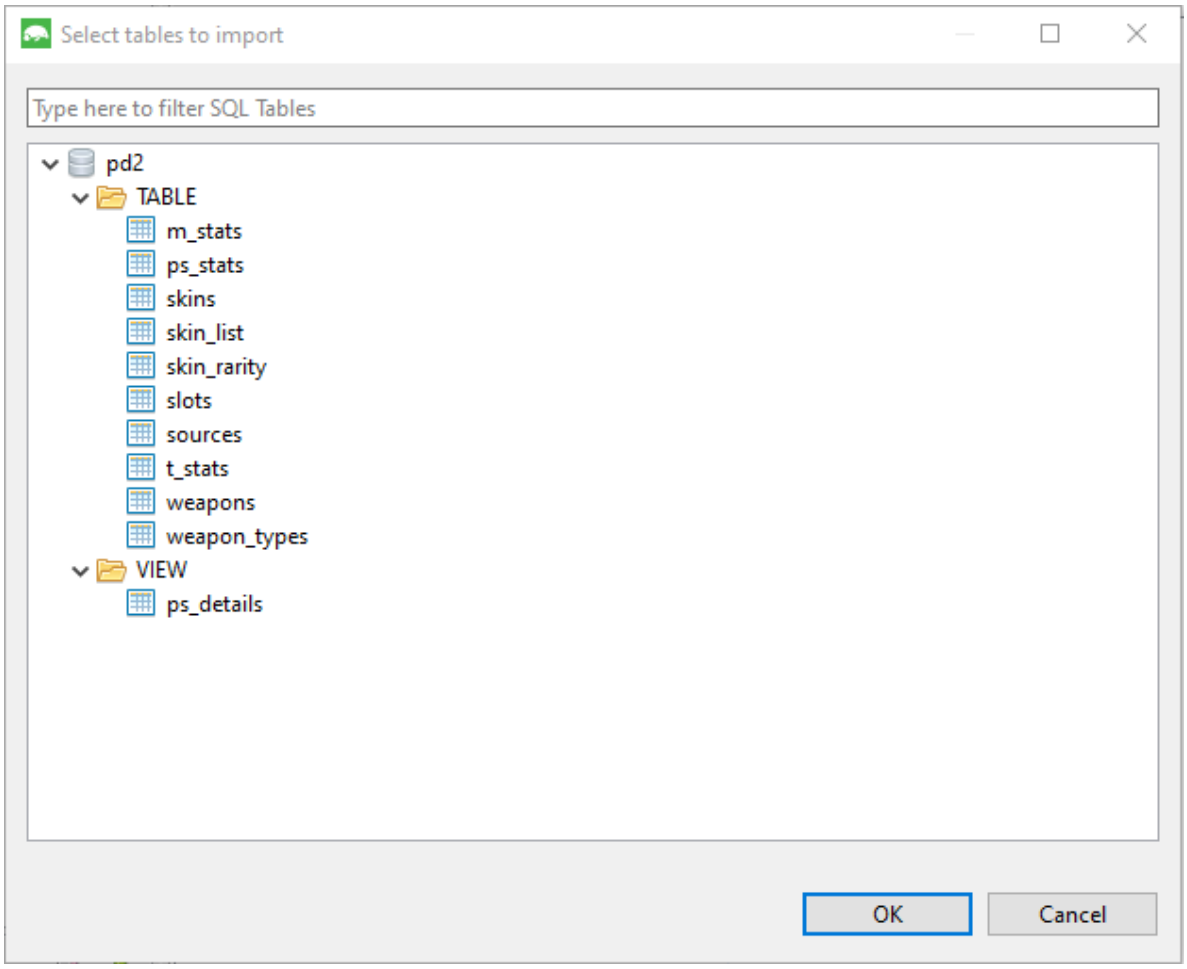


After the two databases were setup, I used the Studio 3T's SQL Migration feature and connected those two databases (Studio 3T 2019). Another way of migrating data one database to another would to manually extract data and then create the NoSQL. We could also do that coding but Potey et al. (2015, p. 8887) suggests that those ways are inefficient.

As you can see the pd2 database has been selected.



We are then able to select all the SQL Tables we want to import. Just to be safe, import all of them and sort them out later.



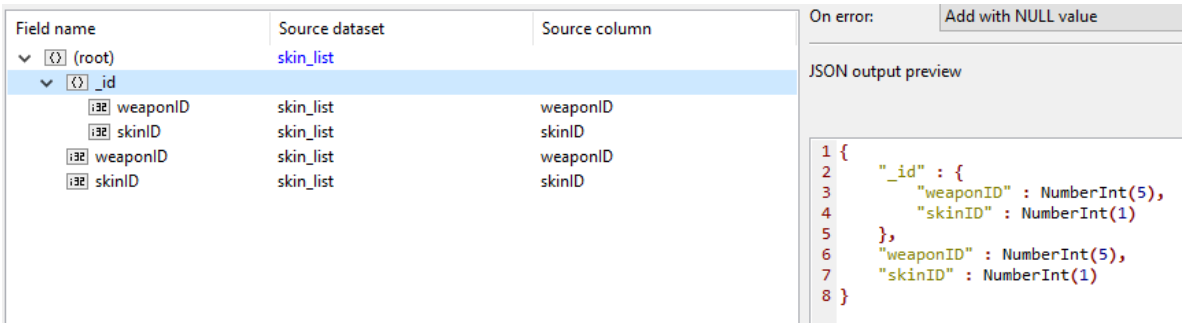
Data Cleaning and Transformation

When the tables were imported from the MySQL server to MongoDB, for some reason the software decided to add an "_id" field to every single table.

Field name	Source dataset	Source column
(root)	skins	
_id	skins	skinID
skinID	skins	skinID
skinName	skins	skinName
sourceID	skins	sourceID
skinRarityID	skins	skinRarityID
damageBonus	skins	damageBonus
accuracyBonus	skins	accuracyBonus
stabilityBonus	skins	stabilityBonus
concealmentBonus	skins	concealmentBonus
totalAmmoBonus	skins	totalAmmoBonus
teamBonus	skins	teamBonus

But each of these tables already had a primary key before they were imported so there is no need for the "_id" field. The first step was to delete the new "_id" for each of these tables before exporting.

The only exception would probably be the weak entity "skin list" since it is the only table that uses a composite key.



Since MongoDB is doesn't have a strict schema there would no longer be any point to having 10 separate tables. There are two main methods available for retaining relationships in MongoDB which are embedding documents and document references. This paper will only be using embedding documents as it is a lot easier to query.

The screenshot below is roughly how each weapon will be roughly represented in JSON format.

```
        "weaponID" : NumberInt(1),
        "rof" : NumberInt(652),
        "totalAmmo" : NumberInt(150),
        "magazine" : NumberInt(30),
        "damage" : NumberInt(50),
        "accuracy" : NumberInt(48),
        "stability" : NumberInt(60),
        "concealment" : NumberInt(16),
        "threat" : NumberInt(14)
    },
    "m_stats" : {

    },
    "t_stats" : {

    },
    "skin_list" : [
        {
            "weaponID" : NumberInt(2),
            "skinID" : NumberInt(28),
            "skins" : [
                {
                    "skinID" : NumberInt(28),
                    "skinName" : "Le Grand Bleu",
                    "sourceID" : NumberInt(8),
                    "skinRarityID" : NumberInt(1),
                    "damageBonus" : NumberInt(0),
                    "accuracyBonus" : NumberInt(4),
                    "stabilityBonus" : NumberInt(0),
                    "concealmentBonus" : NumberInt(0),
                    "totalAmmoBonus" : NumberInt(0),
                    "teamBonus" : 0.0,
                    "skin_rarity" : {
                        "skinRarityID" : NumberInt(1),
                        "skinRarityName" : "Common"
                    },
                    "sources" : {
                        "sourceID" : NumberInt(8),
                        "sourceName" : "Bodhi Safe"
                    }
                }
            ]
        },
```


As you can see for a primary weapon such as this one has "m_stats" and "t_stats" that are for melee and throwables weapons respectively. The next following lines will run in order to remove these empty fields that belong there.

```
db.weapons.update({ps_stats: {}}, {$unset: {"ps_stats":""}}, {multi: true})
db.weapons.update({m_stats: {}}, {$unset: {"m_stats":""}}, {multi: true})
db.weapons.update({t_stats: {}}, {$unset: {"t_stats":""}}, {multi: true})
```

The "skin_list" field was removed for any documents that had no skins

```
db.weapons.update({"skin_list.skins": []}, {$unset: {"skin_list":""}}, {multi:
true})
```

Then the field names of "ps_stats", "m_stats" and "t_stats" were all changed stats to make it easier to query.

```
db.weapons.updateMany({}, { $rename: {'ps_stats':'stats'}})
db.weapons.updateMany({}, { $rename: {'m_stats':'stats'}})
db.weapons.updateMany({}, { $rename: {'t_stats':'stats'}})
```

Results

This is the amount of space the PD2 taken is 304KB the MySQL server. Whereas the PD2 database taken is 237KB. NoSQL databases like MongoDB are clearly more storage efficient compared to relational databases such as MySQL.

This the what the first record looks like now. As you can see the "m_stats" and "t_stats" have been successfully removed.

```
{
    "_id" : NumberInt(1),
    "weaponID" : NumberInt(1),
    "slotID" : NumberInt(1),
    "weaponName" : "AK",
    "weaponTypeID" : NumberInt(2),
    "sourceID" : NumberInt(5),
    "reputation" : NumberInt(1),
    "weapon_types" : {
        "weaponTypeID" : NumberInt(2),
        "weaponTypeName" : "Assault Rifle"
    },
    "slots" : {
        "slotID" : NumberInt(1),
        "slotName" : "Primary"
    },
    "sources" : {
        "sourceID" : NumberInt(5),
        "sourceName" : "Base Game"
    },
    "ps_stats" : {
        "weaponID" : NumberInt(1),
        "rof" : NumberInt(652),
        "totalAmmo" : NumberInt(150),
        "magazine" : NumberInt(30),
        "damage" : NumberInt(50),
        "accuracy" : NumberInt(48),
        "stability" : NumberInt(60),
        "concealment" : NumberInt(16),
        "threat" : NumberInt(14)
    },
    "skin_list" : [
        {
            "weaponID" : NumberInt(2),
            "skinID" : NumberInt(28),
            "skins" : [
                {
                    "skinID" : NumberInt(28),
                    "skinName" : "Le Grand Bleu",
                    "sourceID" : NumberInt(8),
                    "skinRarityID" : NumberInt(1),
                    "damageBonus" : NumberInt(0),
                    "accuracyBonus" : NumberInt(4),
```


Queries

This section demonstrates that the queries that were done in MySQL will continue to work even in MongoDB. All the queries

Accessing information about a primary/secondary weapon

Without restrictions

```
db.weapons.find({}, {
    "name": 1,
    "stats": 1
}).sort({
    "name": 1
});
```

Output:

```
{
    "_id" : NumberInt(1),
    "stats" : {
        "weaponID" : NumberInt(1),
        "rof" : NumberInt(652),
        "totalAmmo" : NumberInt(150),
        "magazine" : NumberInt(30),
        "damage" : NumberInt(50),
        "accuracy" : NumberInt(48),
        "stability" : NumberInt(60),
        "concealment" : NumberInt(16),
        "threat" : NumberInt(14)
    }
}
{
    "_id" : NumberInt(2),
    "stats" : {
        "weaponID" : NumberInt(2),
        "rof" : NumberInt(561),
        "totalAmmo" : NumberInt(90),
        "magazine" : NumberInt(30),
        "damage" : NumberInt(80),
        "accuracy" : NumberInt(60),
        "stability" : NumberInt(44),
        "concealment" : NumberInt(13),
        "threat" : NumberInt(22)
    }
}
{
    "_id" : NumberInt(3),
    "stats" : {
        "weaponID" : NumberInt(3),
        "rof" : NumberInt(652),
        "totalAmmo" : NumberInt(105),
        "magazine" : NumberInt(35),
        "damage" : NumberInt(99),
        "accuracy" : NumberInt(68),
        "stability" : NumberInt(60),
        "concealment" : NumberInt(16),
        "threat" : NumberInt(14)
    }
}
{
    "_id" : NumberInt(4),
    "stats" : {
```


With restrictions

```
db.weapons.find({"weaponID":45}, {
    "name": 1,
    "stats": 1
})
```

Output:

```
{
    "_id" : NumberInt(45),
    "stats" : {
        "weaponID" : NumberInt(45),
        "rof" : NumberInt(333),
        "totalAmmo" : NumberInt(96),
        "magazine" : NumberInt(16),
        "damage" : NumberInt(18),
        "accuracy" : NumberInt(12),
        "stability" : NumberInt(24),
        "concealment" : NumberInt(21),
        "threat" : NumberInt(28)
    }
}
```


Updating a weapon stats

```
db.weapons.find({"weaponID":1})
```


```
    "stats" : {
        "weaponID" : NumberInt(1),
        "rof" : NumberInt(652),
        "totalAmmo" : NumberInt(150),
        "magazine" : NumberInt(30),
        "damage" : NumberInt(50),
        "accuracy" : NumberInt(48),
        "stability" : NumberInt(60),
        "concealment" : NumberInt(16),
        "threat" : NumberInt(14)
    }
}
```


```
db.weapons.updateMany(
    { "weaponID" : 1},
    { $inc: { "stats.damage": 10 }
})
```

```
db.weapons.find({"weaponID":1})
```


```
    "stats" : {
        "weaponID" : NumberInt(1),
        "rof" : NumberInt(652),
        "totalAmmo" : NumberInt(150),
        "magazine" : NumberInt(30),
        "damage" : 60.0,
        "accuracy" : NumberInt(48),
        "stability" : NumberInt(60),
        "concealment" : NumberInt(16),
        "threat" : NumberInt(14)
    }
```


Adding a new field "cost" to one of the documents. Instead of running one query to restructure the table and one to insert the value, MongoDB can do both in one query.

```
db.weapons.updateMany(
    {"weaponID":1},
    { $set: { "cost" : 100 } }
)
```

Querying for weapons without skins

```
db.weapons.find({"skin_list.skins":{"$exists":false}},
    {"weaponName": 1})
```

```
{
    "_id" : NumberInt(121),
    "weaponName" : "Bolt Cutters"
}
{
    "_id" : NumberInt(122),
    "weaponName" : "Money Bundle"
}
{
    "_id" : NumberInt(123),
    "weaponName" : "Lucille Baseball Bat"
}
{
    "_id" : NumberInt(124),
    "weaponName" : "Rivertown Glen Bottle"
}
{
    "_id" : NumberInt(125),
    "weaponName" : "Alpha Mauler"
}
{
    "_id" : NumberInt(126),
    "weaponName" : "K.L.A.S. Shovel"
}
{
    "_id" : NumberInt(127),
    "weaponName" : "Telescopic Baton"
}
{
    "_id" : NumberInt(128),
    "weaponName" : "Survival Tomahawk"
}
{
    "_id" : NumberInt(129),
    "weaponName" : "Utility Machete"
}
{
    "_id" : NumberInt(130),
    "weaponName" : "Compact Hatchet"
}
{
    "_id" : NumberInt(131),
    "weaponName" : "Ding Dong Breaching Tool"
}
{
    "_id" : NumberInt(132),
    "weaponName" : "Baseball Bat"
```


Querying for the most damage weapon

```
db.weapons.find().sort({"stats.damage":-1}).limit(1)
```


```
//Output:
{
    "_id" : NumberInt(116),
    "weaponID" : NumberInt(116),
    "slotID" : NumberInt(2),
    "weaponName" : "HRL-7",
    "weaponTypeID" : NumberInt(6),
    "sourceID" : NumberInt(62),
    "reputation" : NumberInt(38),
    "weapon_types" : {
        "weaponTypeID" : NumberInt(6),
        "weaponTypeName" : "Special"
    },
```

```
    "slots" : {
        "slotID" : NumberInt(2),
        "slotName" : "Secondary"
    },
    "sources" : {
        "sourceID" : NumberInt(62),
        "sourceName" : "The OVERKILL Pack"
    },
    "skin_list" : {
        "weaponID" : NumberInt(116),
        "skinID" : NumberInt(502),
        "skins" : [
            {
                "skinID" : NumberInt(502),
                "skinName" : "Headline",
                "sourceID" : NumberInt(51),
                "skinRarityID" : NumberInt(1),
                "damageBonus" : NumberInt(0),
                "accuracyBonus" : NumberInt(4),
                "stabilityBonus" : NumberInt(0),
                "concealmentBonus" : NumberInt(0),
                "totalAmmoBonus" : NumberInt(0),
                "teamBonus" : 0.0,
                "skin_rarity" : {
                    "skinRarityID" : NumberInt(1),
                    "skinRarityName" : "Common"
                },
                "sources" : {
                    "sourceID" : NumberInt(51),
                    "sourceName" : "Sputnik Safe"
                }
            }
        ]
    },
    "stats" : {
        "weaponID" : NumberInt(116),
        "rof" : NumberInt(30),
        "totalAmmo" : NumberInt(4),
        "magazine" : NumberInt(1),
        "damage" : NumberInt(10000),
        "accuracy" : NumberInt(96),
        "stability" : NumberInt(96),
        "concealment" : NumberInt(5),
        "threat" : NumberInt(37)
    }
}
```


Counting the number of each type of weapon

```
db.weapons.aggregate([
    {"$group": {_id:"$weapon_types.weaponTypeName", count:{$sum:1}}},
    {"$sort":{"count":-1}}
])
```

Output (in tabular form):

_id	count
Axe	37.0
Assault Rifle	24.0
Knife	22.0
Submachine G...	17.0
Pistol	17.0
Special	16.0
Shotgun	15.0
Akimbo	14.0
Sniper Rifle	10.0
Fists	5.0
Light Machine ...	5.0
Projectile	5.0
Fragmentation ...	3.0
Weapon	2.0
Hypodermic Sti...	1.0
Sword	1.0
Alcohol Flask	1.0
Improvised Inc...	1.0

Limitation

There doesn't seem to be a way to get rid of the weak entity. An example of this is how the "skins" are inside the "skin_list", even though the "skin_list" is no longer needed.

Although embedding documents is very efficient and appropriate in this case, it has its drawbacks. For example, we have users and roles, which have nothing to do with the database that has been migrated. We can embed user documents inside role documents or the other around. If we decided that users are the more important entity and roles should be embedded within them, it would look something like this:

```
Users:
[
   {
       id: 1234,
      email: 'bob@admin.com',
      role: {
         name: 'Admin',
         description: 'A user with awesome powers.'
      }
   },
   {
       id: 4321,
      email: 'bill@admin.com',
      role: {
         name: 'Admin',
         description: 'A user with awesome powers.'
      }
   }
]
```

However, if we were to update the description of the "Admin" role, it would only be applied to the first user and not the second user. We could attempt to fix this issue by either going through each document, check if the role is "Admin" and change the description, which is not very scalable in a larger database. The other option would be to embed users within roles instead, which would look something like this [15]:

```
Roles:
[
   {
       id: 5678,
      name: 'Admin',
      description: 'A user with awesome powers.',
      users: [
         {
            email: 'bob@admin.com'
         },
         {
            email: 'bill@admin.com'
         }
      ]
   }
]
```

This solution is a much more viable, however, if we were to introduce a "Teams" entity that contains multiple users, we would have the same problem.

Conclusion

Overall the transition MySQL to MongoDB was a manageable learning curve. Most of the time was cleaning the tables before they got imported into MongoDB and learning the query language for MongoDB. Choosing to embed documents made it possible to retrieve all the related data with just one query, which is not have been possible with document references. Embedding documents definitely had its downside which may be alleviated with document references, which would be another topic to explore. It became apparent that the database sourced from the relational database server should have been implemented in NoSQL in the first place. If it was written in NoSQL there would not have been anything need for Views or weak entities, reducing storage space. While NoSQL worked well with this database, that will not always be the case.

References

Abramova, V & Bernardino, J 2013, NoSQL databases: MongoDB vs cassandra. In *Proceedings of the international C**** *conference on computer science and software engineering* (pp. 14-22). ACM.

Babu, A & Surendran, S 2017, Relational to NoSQL Database Migration. *International Journal of Innovative Research in Science, Engineering and Technology. ISSN (Online)*, pp.2319-8753.

DB-Engines 2019, *MongoDB System Properties*, viewed 29 October 2019, <https://db-engines.com/en/system/MongoDB>.

DB-Engines 2019, *MySQL System Properties*, viewed 29 October 2019, <https://db-engines.com/en/system/MongoDB>.

Headley, J 2019, *The Problem with MongoDB*, viewed 29 October, 2019, <https://hackernoon.com/the-problem-with-mongodb-d255e897b4b>.

MongoDB 2019, *FAQ: MongoDB Fundamentals — MongoDB Manual*, viewed 29 October, 2019, <https://docs.mongodb.com/manual/faq/fundamentals/>.

MongoDB 2019, *NoSQL vs SQL (Relational Databases)*, viewed 29 October 2019, <https://www.mongodb.com/scale/nosql-vs-relational-databases>.

MongoDB 2019, *Types of NoSQL Databases*, viewed 29 October, 2019, <https://www.mongodb.com/scale/types-of-nosql-databases>.

Potey, M, Digrase, M, Deshmukh, G & Nerkar, M 2015, Database migration from structured database to non-structured database. *International Journal of Computer Applications*, *975*, p.8887.

Sasaki, B.M 2018, *Graph Databases for Beginners: ACID vs. BASE Explained*, neo4j, viewed 29 October, 2019, <https://neo4j.com/blog/acid-vs-base-consistency-models-explained/>.

Studio 3T 2019, *SQL to MongoDB Migration*, viewed 29 October 2019,<https://studio3t.com/knowledge-base/articles/sql-to-mongodb-migration/>.

Tang, E & Fan, Y 2016, November. Performance comparison between five NoSQL databases. In *2016 7th International Conference on Cloud Computing and Big Data (CCBD)* (pp. 105-109). IEEE.

Yoo, J, Lee, K.H & Jeon, Y.H, 2018, Migration from RDBMS to NoSQL Using Column-Level Denormalization and Atomic Aggregates. *Journal of Information Science & Engineering*, *34*(1).