TASK 1 PART 2

## Database Choice:

Doing a quick analysis of the database, we can see that there is mainly only one table, where all fields directly relate to the ID of that row. We can view the database as a set of IDs(distinct), where we associate with each ID all the values from that particular row. Therefore, there are no hard relations to implement (no foreign keys). As so, an RDBMS wouldn’t make much sense. Having this in mind, I chose MongoDB as the platform for implementing this database.

# How to store and query:

MongoDB is a document-based database, meaning that each entry(row) represents a set of key-value pairs (where key would be the column name, value would be the value of that column to that ID). In case one of the values has multiple elements, then the value will become a list. If the value is an object (maybe a row inside another table), MongoDB supports embedded documents, meaning that the value of a key-value pair inside a document can be another document. This gives a lot of flexibility to MongoDB, plus we don’t have to specify any data types and not all documents have to contain all keys. If we don’t know the temperature of a certain distant planet, then we can just not include the key temperature inside that planet’s document. In RDBMS, you have to hard type each column, which means you can have very frustrating errors when inserting rows, especially when you have over 300 columns. In MongoDB, types don’t have to be specified. It is true, if you have to run aggregate functions like AVG, you need numeric types, however the smart querying of MongoDB and the fact you don’t have to alter the structure of your tables, means you can very easily update types, as I will prove below.

To use mongodb, you need to have a mongod instance running. MongoDB easiliy integrates with various programming languages, in this case I will use Python.

We start by downloading the .csv file from the link and then we will use Python to store the values in an array:

**import** csv

f=open('c:/Student/220CT/planets.csv','r')

reader=csv.DictReader(filter(**lambda** row: row[0]!="#",f)) #eliminate all comments

l=list(reader)

f.close()

#convert all numeric values to floats - proves how easy it is to do it

#with mongoDB compared to an RDBMS

**for** doc **in** l:

**for** key **in** doc.keys():

**try**:

doc[key]=float(doc[key])

**except** ValueError:

**continue**

######################################################################

Now that we have our values, we will now open mongoDB, create a database called exoplanets, in which we will create a collection exoplanets where we will store all our values:

**import** pymongo

client=pymongo.MongoClient("localhost",27017)

db=client.exoplanets

results = db.exoplanets.insert\_many(l)

Now all our values are stored. If we want to further introduce new values, mongoDB easily supports that by using db.collection.insert\_one(document), or insert\_many for bulk insertion. The best thing is that there are no rules to what your documents can hold, so you can add extra values to any document if that is what you need.

You can find an entry with the command db.collection.find\_one(matching query). This returns a cursor to that document.

Let’s test some queries on our new collection:

#Prints all planets found by Imaging, prints name, jmass, eqt and tradur

cursor=db.exoplanets.find(

{"pl\_discmethod":"Imaging"})

**for** doc **in** cursor:

**print**("name: " + doc["pl\_hostname"],end=" ")

**print**("jupiter\_mass: " +str(doc["pl\_massj"]),end=" ")

**print**("pl\_eqt: "+str(doc["pl\_eqt"]),end=" ")

**print**("pl\_trandur: "+str(doc["pl\_trandur"]),end=" ")

**print**()

Output:

name: 1RXS J160929.1-210524 jupiter\_mass: 8.0 pl\_eqt: 1700.0 pl\_trandur:

name: 2MASS J01225093-2439505 jupiter\_mass: 24.5 pl\_eqt: pl\_trandur:

name: 2MASS J02192210-3925225 jupiter\_mass: 13.9 pl\_eqt: pl\_trandur:

name: 2MASS J04414489+2301513 jupiter\_mass: 7.5 pl\_eqt: pl\_trandur:

name: 2MASS J12073346-3932539 jupiter\_mass: 4.0 pl\_eqt: 1150.0 pl\_trandur:

name: 2MASS J21402931+1625183 A jupiter\_mass: 20.95 pl\_eqt: 2075.0 pl\_trandur:

name: 2MASS J22362452+4751425 jupiter\_mass: 12.5 pl\_eqt: pl\_trandur:

name: 51 Eri jupiter\_mass: 2.0 pl\_eqt: 700.0 pl\_trandur:

name: AB Pic jupiter\_mass: 13.5 pl\_eqt: pl\_trandur:

name: CFBDSIR J145829+101343 jupiter\_mass: 10.5 pl\_eqt: 370.0 pl\_trandur:

name: CHXR 73 jupiter\_mass: 12.569 pl\_eqt: pl\_trandur:

name: CT Cha jupiter\_mass: 17.0 pl\_eqt: 2600.0 pl\_trandur:

etc. (many more)

We do have the structure of the table inside our csv file, so we can rename our columns from those weird names to more meaningful ones based on the documentation. As we can see, some of the documents don’t have some of those values, but this is not an issue to the structure of the database.

One more:

#prints the average eqt for all planets, grouped by method of discovery

cursor=db.exoplanets.aggregate([

{"$match": {"pl\_eqt": {"$exists":True,"$ne":""}}},

{"$group": {"\_id":"$pl\_discmethod", "avg\_temp": {"$avg": "$pl\_eqt"}}}

])

**for** doc **in** cursor:

**print**(doc)

Output:

{'avg\_temp': 2140.0, '\_id': 'Orbital Brightness Modulation'}

{'avg\_temp': 455.0, '\_id': 'Transit Timing Variations'}

{'avg\_temp': 494.8181818181818, '\_id': 'Radial Velocity'}

{'avg\_temp': 1260.1120943952803, '\_id': 'Transit'}

{'avg\_temp': 69.33333333333333, '\_id': 'Microlensing'}

{'avg\_temp': 1502.6818181818182, '\_id': 'Imaging'}

This one prints the average temperature of all planets, grouped by methods of discovery, and ignores those entries that don’t have a temperature or have an invalid entry. This proves how flexible mongoDB is with handling large data and errors.

# Benefits of the MongoDB database solution

Several benefits have already been explored. We can insert data in an unstructured way, meaning we can have different keys in each documents, and can have different data types for each key as well. The data stored in MongoDB can in fact have any kind of structure, any format and come from any source.

The operations done on the database are done in real-time, because MongoDB can directly analyse the structure of the data within the database. MongoDB offers great scalability through the fact that it stores the data in an HDMS style, in clusters. Traditional RDBMS can come with many challenges when you attempt to cluster them. In MongoDB it is very easy. Sharding is the process of splitting the data on multiple machines, meaning that a mongoDB database scales horizontally, not vertically(more data, more commodity cheap hardware). You can specify how you want to shard your data, through a specific key, or compound key(in this case, we could split by temperature, create as many buckets as we want(shards)). Through this process, you can hold trillions of records at a relatively cheap cost.

MongoDB offers a replication process which secures your data. If a particular node that holds data fails, mongoDB ensures that the data is still available, by replicating it whenever it is created or updated. Usually 3 copies of the same data are kept inside an HDMS cluster.

Consistency is also achieved by updating the copies of the data in real time after a write. Strong consistency can slow down a system if you perform much more writes than reads, but if you perform much more read than writes, strong consistency is a good choice.

MongoDB also offers high availability. If the master node fails, then all other nodes will elect a new master node, ensuring the continuity of your database manipulation.

Given the fact the database currently holds over 3000 rows, with over 300 columns each, the sheer volume of the data would be hard to manage in an RDBMS. The solution presented above will resolve all the challenges expected from such a big database. It is always available, your data is secured through replication, it is easily scalable, very fast and easy to use.