

NORWEGIAN UNIVERSITY OF SCIENCE AND TECHNOLOGY

DEPARTMENT OF CHEMICAL ENGINEERING

ADVANCED PROCESS SIMULATION

SQL vs. NoSQL

Author: Cansu Birgen

Supervisors:

Prof. Heinz Preisig

John Morud

December 8, 2014

Abstract

Chemical databases have been expanding rapidly both in complexity and amount. Traditional SQL approach for chemical databases has not been sufficient to meet changing requirements in data storage and retrieval. Flexible and schemaless data model of NoSQL data store MongoDB provides the opportunity for storage of any type of data which can be inserted by several different users without imposing a certain data structure. A chemical database developed in MongoDB is analyzed to investigate limitations and advantages in a comparative manner with SQL approach. Basic read and write operations are performed as well as data storage aspects are studied. Flexibility and horizontal scaling of MongoDB increase its desirability while its immaturity causes consistency and security issues acting as obstacles to its wider deployment.

Contents

1. Aim and Scope.....	1
2. Introduction.....	1
2.1. Database Management Systems	1
3. MongoDB.....	8
3.1. MongoDB Terminology and Concepts	8
3.2. MongoDB CRUD Operations.....	9
3.3. MongoDB Aggregation	13
3.4. MongoDB Indexing	16
4. MongoChem.....	16
4.1. Data Exploration.....	17
4.2. Data Query and Modification	19
5. Pymatgen.....	24
5.1. Pymatgen Examples.....	25
6. Discussion and Conclusions	28
References.....	30
Appendix.....	32

Figure 1 Representation of relational database model (source: wikipedia)	2
Figure 2 A representation of key-value stores	6
Figure 3. A representation of column stores	6
Figure 4 An illustration of graph databases	7
Figure 5 An illustration of document stores	7
Figure 6 Electronic structure of Fe atom	27

Table 1 ACID and BASE comparative representation (Vanroose et. Thillo, 2014).....	5
Table 2 Classification and comparison of NoSQL databases (Popescu, 2010)	8
Table 3 Terminology and concepts in SQL and MongoDB (MongoDB manual).....	9
Table 4 SQL statements related to table-level actions and the corresponding MongoDB statements	10
Table 5 SQL statements related to inserting records into tables and the corresponding MongoDB statements.....	11
Table 6 SQL statements related to reading records from tables and the corresponding MongoDB statements.....	11
Table 7 SQL statements related to updating existing records in tables and the corresponding MongoDB statements.....	13
Table 8 SQL statements related to deleting records from tables and the corresponding MongoDB statements.....	13
Table 9 SQL aggregation terms, functions, and concepts and the corresponding MongoDB aggregation operators.....	13
Table 10 SQL aggregation statements and the corresponding MongoDB statements	14

1. Aim and Scope

Aim of this study is to investigate non-relational database approach, specifically MongoDB in comparison with relational database approach in terms of data storage, organization and manipulation applications for chemical databases. The study aims to provide insight in compatibility, advantages and flaws for core database operations.

Document-based store, MongoDB is analyzed within the context of chemical data storage and retrieval. Basic operations are performed by using Mongo Query Language to unleash properties of a chemical database MongoChem as the application of NoSQL approach in comparison with SQL approach. Pymatgen, a Python library which uses MongoDB for data storage is studied to illustrate a more comprehensive deployment of MongoDB. Comparison evolves around the common chemical database operations and data models supported with perspectives from literature review about general characteristics of MongoDB.

2. Introduction

Scientific fields such as chemistry and material science are experiencing rapid growth in data volumes, data variety, and the rate at which data needs to be analyzed. Exploratory data analysis, where scientists use data mining and statistical techniques to search for patterns, is difficult at this scale with currently available tools. New tools are needed to handle the large amounts of semi-structured and/or structured scientific data. Scalable methods for both simple queries as well as complex analysis are necessary (Dede et. al., 2013).

2.1. Database Management Systems

A database can basically be defined as an organized collection of data which enables us to handle large quantities of information by inputting, storing, retrieving and managing them. A database management system (DBMS) is defined as a suite of computer software which provides the interface between users and a database. Maintenance of integrity and security of the data stored in the database as well as the recovery of the information in case of a system fail are the duties of a DBMS.

There are different classification criteria for databases and RDBMSs. Categorization can be done according to the database model that they support such as relational, the type of computer they run on which can vary from a server cluster to a mobile phone, the query language used to access the database such as SQL, and their internal engineering, which has an impact on crucial characteristics such as performance, scalability, resilience, and security.

Within the scope of this study, the database model is taken into account and studied more in detail. Two major categories which are relational database and non-relational database are studied in a comparative manner.

2.1.1. Relational Approach

A relational database is defined as a database in which the data is organized based on the relational model of data (Codd, 1970). The purpose of this model is to provide a declarative method for data and query specification.

In relational database model, data is represented as rectangular tables which are known as relations. A relation has a set of named attributes. These attributes can be considered as names of the table columns. Each attribute is associated with a domain, i.e. an atomic type such as an integer or a string. A row in a table is called a tuple. When a relation has N attributes, each tuple contains N components (one for each attribute). Every component must belong to the domain of its corresponding attribute. A schema is the name of a relation together with its set of attributes (Näsholm, 2012). A generic representation of relational database model can be seen in Figure 1.

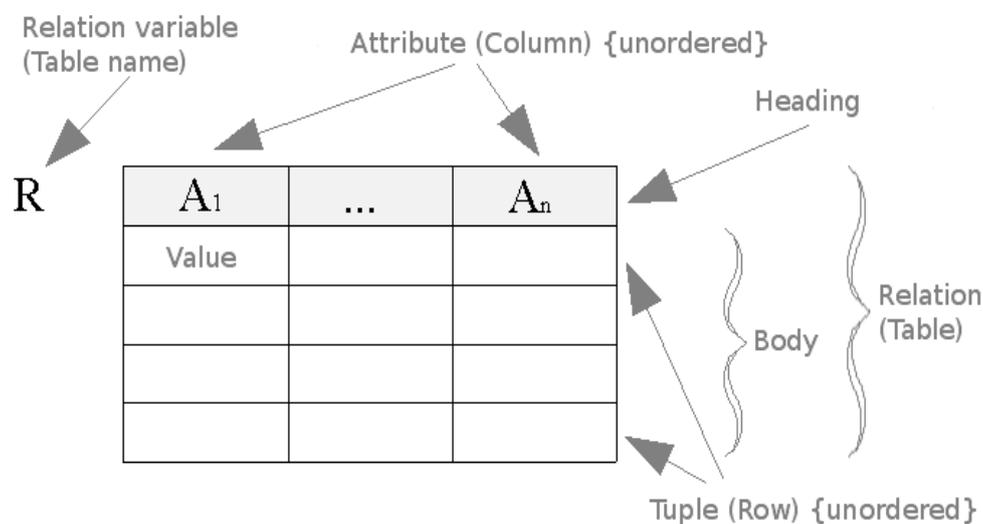


Figure 1 Representation of relational database model (source: wikipedia)

Most RDBMSs guarantee so called ACID transactions. ACID is an acronym of four properties. ACID (atomicity, consistency, isolation, durability) stands for a set of properties desired for reliable database transactions. ACID transactions provide the following assurances (Pritchett, 2008):

- Atomicity: All of the operations in the transaction will complete, or none will.
- Consistency: The database will be in a consistent state when the transaction begins and ends.
- Isolation: The transaction will behave as if it is the only operation being performed upon the database.
- Durability: Upon completion of the transaction, the operation will not be reversed.

Relational databases mostly use structured query language (SQL). Data insert, query, update and delete, schema creation and modification, and data access control are included in the scope of SQL.

It is specifically designed for querying the data contained in a relational database. SQL is a set-based, declarative query language.

2.1.2. Non-relational Approach (NoSQL)

Non-relational databases are named as NoSQL (Not Only SQL) which provides a mechanism for storage and retrieval of data which is modeled in a way different than in a relational database. NoSQL emphasizes the movement coming up with alternatives for RDBMSs/SQL where these are a bad fit rather than being being completely against them. Therefore, NoSQL is not meant to be a total replacement for relational approach. However, there is no still prescriptive definition of NoSQL. It consists of aggregation of common characteristics (Näsholm, 2012).

The term NoSQL dates back to 1998 when it was used for a particular RDBMS that did not support SQL. It was not until 2009 that it was used with approximately the same meaning that it has today.

Evolution of NoSQL databases was initiated by the need of a data storage model which enables the users work with large volumes of data with database running on clusters, since relational databases are not designed to run efficiently on clusters (Fowler et. Pramod, 2012). Common characteristics of NoSQL databases are shown below.

- Not using the relational model
- Running well on clusters
- Open-source
- Built for 21st century web estates
- Schemaless

After general introduction of NoSQL, a more detailed background will be provided in the following subsections.

Motives for Deployment of NoSQL

For a long time, RDBMSs have been the solution in many applications of database management. However, some other applications have faced changing requirements, and RDBMSs were not fully capable of meeting the requirements (Strauch, 2011). This situation created a demand for a new approach in database management thus triggered the interest in NoSQL. Some of the important features which have expanded the motives for deployment of NoSQL are listed below.

- For some applications, it is necessary to process huge amount of data. NoSQL handles it better compared to many RDBMSs.
- RDBMSs were originally built to scale vertically referring to adding more hardware onto existing machines. However, vertical scaling has some limitations as it gives sublinear effects, sometimes requires high-risk operational efforts and there is an upper bound on how

much hardware that can be added and efficiently utilized by the software. These limitations are in favor of NoSQL approach, since it is able to scale horizontally which stands for adding more machines to a cluster. Horizontal scaling does not suffer from those limitations (Strauch, 2011). Horizontal scalability has also been offered by some RDBMSs; however, its utilization is not straight-forward (Näsholm, 2012).

- Currently, object-oriented programming is the mostly widely practiced programming paradigm. In RDBMSs, persisting applications objects require a mapping between the object model of the application and relational database model of the database. Creating this demands time and effort, and requires the data to be treated relationally although it is not the case inherently.
- Development of RDBMSs has imposed the idea of 'One Size Fits All' meaning that they are seen as a generic tool for handling all kinds of requirements on data management. This idea has been criticized as RDBMSs were too much at once which causes in excelling at nothing (Stonebraker et. al., 2007). Different requirements in terms of consistency, performance and so on can be observed in different applications. Therefore, 'One Size Fits All' concept has flaws.
- Most RDBMSs provide ACID transactions. Sacrificing from ACID might result in trade-offs. In some applications, ACID is not needed therefore; there will not be any loss when it is not provided (Näsholm, 2012).
- As open source projects, several NoSQL databases can be used for free. Many RDBMS alternatives are in comparison rather expensive. Also, because of less rigid data models and no explicitly defined schema, application development is often faster using NoSQL databases, resulting in lower development costs (Näsholm, 2012).

Principle NoSQL Concepts

There are some basic concepts employed in NoSQL model. Since NoSQL is still a broad concept, there are exceptions for almost all the characteristics written below.

- NoSQL databases are mostly distributed systems in which several machines work together in clusters to provide data. To achieve redundancy and high availability, each data piece is replicated over those machines.
- Horizontal scalability of NoSQL is a major advantage. It enables the user to add nodes dynamically to a cluster without any downtime which gives linear effects on storage and overall processing capacities. There is no upper limit for number of machines that can be added.
- Huge amount of data can easily and quickly be handled by NoSQL since they are built in that way.
- Structure of NoSQL data is not defined explicitly with schema introduced in the database.

Instead of this, clients can store data as they wish without depending on a predefined schema/structure.

- NoSQL can employ non-relational data models which allow for more complex and sophisticated structures.
- RDBMSs mostly support SQL while NoSQL generally do not. Each NoSQL system has its own query interface. There have been attempts to unify these interfaces.
- CAP (consistency, availability and partition tolerance) theorem refers the three fundamental properties of shared-data systems which are data consistency, system availability and tolerance to network partitions (Brewer, 2000). In short, the theorem indicates that at most two of those three properties can be presented for any shared data system. This statement applies to relational databases and favors NoSQL databases. However, most of the NoSQL stores compromise consistency in favor of availability and partition tolerance (Pritchett, 2008). If a system or parts of a system have to be consistent and partition-tolerant, ACID properties are required. If availability and partition-tolerance are favored over consistency, the resulting system such as NoSQL can be characterized by the BASE (Basically Available, Soft-state, Eventual consistency) properties. Thus, most of the NoSQL stores lack true ACID transactions (Vanroose et. Thillo, 2014). This trade-off has been a critical discussion point when principle concepts of databases are analyzed. Both sets of properties (ACID and BASE) are summarized and represented in Table 1. As can be seen in the table, for BASE databases, temporary database inconsistencies are accepted resulting in faster responses in a more scalable manner (Vanroose et. Thillo, 2014).

Table 1 ACID and BASE comparative representation (Vanroose et. Thillo, 2014)

ACID (relational)	BASE (NoSQL)
Strong consistency	Weak consistency
Isolation	Last write wins
Transaction	Program managed
Robust database	Simple database
Simple code (SQL)	Complex code
Available and consistent	Available and partition-tolerant
Scale-up (limited)	Scale-out (unlimited)
Shared (disk, mem, proc etc.)	Nothing shared (parallellizable)

NoSQL Classification

There are different approaches for classification of NoSQL databases resulting in different categories and subcategories (Shermin, 2013). However, most generic and commonly used classification was proposed by Ben Scofield in his presentation at codemesh (2010) which involved a brief comparison of different NoSQL database categories and relational databases (Scofield,

2010). They can be briefly described as follows.

Key-Value Stores: Their fundamental data model is associative array. In this model, data representation is in form of a collection of the key-value pairs in way that each possible key appears only once in the collection. Examples include Berkeley DB, Oracle NoSQL, LevelDB, Dynamo, Memcached. A representation of key-value stores can be seen in Figure 2.

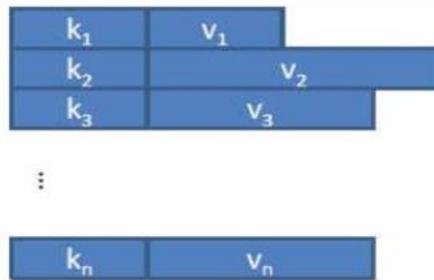


Figure 2 A representation of key-value stores

Column Stores: They store data tables as sections of columns of data. It consists of a unique name, value and timestamp. Examples include Google Bigtable (2006), HBase, Hypertable, Cassandra. A representation of column stores can be seen in Figure 3

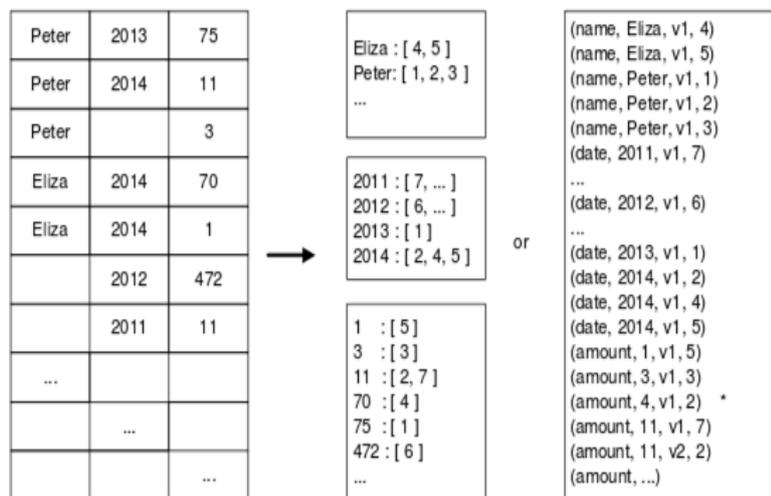


Figure 3. A representation of column stores

Graph Databases: They use graph structures based on graph theory and employ nodes, edges and properties to represent and store data. Every element contains a direct pointer based on system id's to its adjacent elements and no index lookups are required. Examples include Neo4j, InfoGrid, IMS. An illustration of graph databases can be seen in Figure 4.

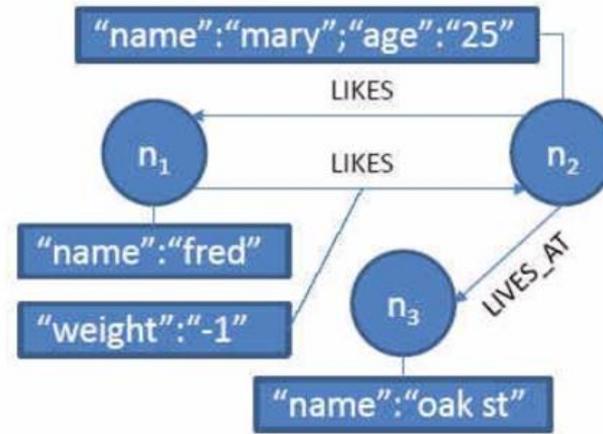


Figure 4 An illustration of graph databases

Document Stores: They are designed around an abstract idea of a document which contain vast amount of data. Document stores accept documents in a variety of forms, and encapsulate them in a standardized internal format. Moreover, they provide support for lists, pointers and nested documents. The requests can be expressed in terms of key or attribute if index exists. Examples include MongoDB, CouchDB, RaptorDB, Riak, IBM Lotus Notes. An illustration of document stores can be seen in Figure 5; k_1, k_2, \dots, k_n are the keys and the corresponding information are their values.

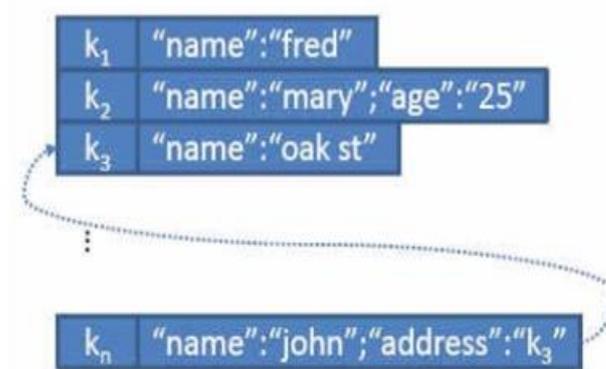


Figure 5 An illustration of document stores

Scofield's ideas were summarized in Popescu's blog which can be seen in Table 2 (Popescu, 2010).

Table 2 Classification and comparison of NoSQL databases (Popescu, 2010)

	Performance	Scalability	Flexibility	Complexity	Functionality
Key-Value Stores	high	high	high	none	variable (none)
Column Stores	high	high	moderate	low	minimal
Document Stores	high	variable (high)	high	low	variable (low)
Graph Databases	variable	variable	high	high	graph theory
Relational Databases	variable	variable	low	moderate	relational algebra

Within the scope of this study, Document Store category is taken into consideration.

3. MongoDB

MongoDB is a schemaless document store database written in C++ and developed in an open-source project by the company 10gen Inc. The motivation behind its development is to close the gap between the fast and highly scalable key-value- stores and feature-rich traditional RDBMSs (Shermin, 2013).

3.1. MongoDB Terminology and Concepts

Some fundamental features of MongoDB are shown below to provide more concrete insight and detailed information about the terminology and concepts.

Document database: A record in MongoDB is a document which is a data structure composed of field and value pairs. The values of fields can also include other documents, arrays, and arrays of documents. MongoDB documents are in BSON ('Binary JSON') data format. Essentially, it is a binary form for representing objects or documents. Using document is advantageous since the documents correspond to native data types in many programming languages, embedded documents and arrays reduce need for expensive joins, and dynamic schema supports fluent polymorphis (MongoDB manual). In the documents, the value of a field can be any of the BSON data types, including other documents, arrays, and arrays of documents. MongoDB stores all documents in collections. A collection is a group of related documents that have a set of shared common indexes. Collections are analogous to a table in relational databases.

High performance: MongoDB provides high performance data persistence. In particular, support for embedded data models reduces I/O activity on database system, and indexes support faster queries and can include keys from embedded documents and arrays (MongoDB manual).

High availability: To provide high availability, MongoDB’s replication facility, called replica sets, provide automatic failover and data redundancy. A replica set is a group of MongoDB servers that maintain the same data set, providing redundancy and increasing data availability (MongoDB manual).

Automatic scaling: MongoDB provides horizontal scalability as part of its core functionality. Automatic sharding distributes data across a cluster of machines. Replica sets can provide eventually-consistent reads for low-latency high throughput deployment (MongoDB manual).

Since the aim of this study to provide insight in MongoDB in a comparative manner with more widely used relational database (SQL) approach, terminology and concepts of these two are summarized and shown in Table 3.

Table 3 Terminology and concepts in SQL and MongoDB (MongoDB manual)

SQL	MongoDB
Database	Database
Table	Collection
Row	document or BSON document
Column	Field
Index	Index
table joins	embedded documents and linking
primary key (specify any unique column or column combinations as primary key)	primary key (the primary key is automatically set to the <code>_id</code> field in MongoDB)
aggregation (e.g. by group)	aggregation pipeline

3.2. MongoDB CRUD Operations

CRUD (create, read, update, delete) stands for the four basic functions of data storage, initially defined for SQL. For MongoDB, these are represented as query which corresponds to read operations while data modification stands for create, update and delete operations.

In MongoDB, read operation is a query which targets a specific collection of documents. Queries specify criteria, or conditions, that identify the documents that MongoDB returns to the clients. A query may include a projection that specifies the fields from the matching documents to return.

- MongoDB queries exhibit the following behavior:
- All queries in MongoDB address a single collection.
- Queries can be modified to impose limits, skips, and sort orders.

- The order of documents returned by a query is not defined unless a sort() method is used.
- Operations that modify existing documents use the same query syntax as queries to select documents to update.
- In aggregation pipeline, the \$match pipeline stage provides access to MongoDB queries.

Create, update and delete operations are represented as data modification operations in MongoDB. They modify the data of a single collection. There are three classes of data modification operations in MongoDB: insert , update , and remove.

Insert operations add new data to a collection. Update operations modify existing data, and remove operations delete data from a collection. No insert, update, or remove can affect more than one document atomically. For the update and remove operations, criteria, or conditions can be set that identify the documents to update or remove. These operations use the same query syntax to specify the criteria as read operations.

The following tables present the various SQL statements and the corresponding MongoDB statements. The examples in the table assume the following conditions:

- The SQL examples assume a table named users.
- The MongoDB examples assume a collection named users that contain documents of the following prototype:

```
{
  _id: ObjectId("509a8fb2f3f4948bd2f983a0"),
  user_id: "abc123",
  age: 55,
  status: 'A'
}
```

Table 4 SQL statements related to table-level actions and the corresponding MongoDB statements

SQL Schema Statements	MongoDB Schema Statements
CREATE TABLE users (id MEDIUMINT NOT NULL AUTO_INCREMENT, user_id Varchar(30), age Number, status char(1),	db.createCollection("users") or db.users.insert({ user_id: "abc123", age: 55, status: "A" })

PRIMARY KEY (id))	
ALTER TABLE users ADD join_date DATETIME	db.users.update({ }, { \$set: { join_date: new Date() } }, { multi: true })
ALTER TABLE users DROP COLUMN join_date	Collections do not describe or enforce the structure of its documents; Still one can use \$unset – db.users.update({ }, { \$unset: { join_date: "" } }, { multi: true })
CREATE INDEX idx_user_id_asc ON users(user_id)	db.users.ensureIndex({ user_id: 1 })
CREATE INDEX idx_user_id_asc_age_desc ON users(user_id, age DESC)	db.users.ensureIndex({ user_id: 1, age: -1 })
DROP TABLE users	db.users.drop()

Table 5 SQL statements related to inserting records into tables and the corresponding MongoDB statements

SQL INSERT Statements	MongoDB insert() Statements
INSERT INTO users(user_id, age, status) VALUES("bcd001", 45, "A")	db.users.insert({ user_id: "bcd001", age: 45, status: "A" })

Table 6 SQL statements related to reading records from tables and the corresponding MongoDB statements

SQL SELECT Statements	MongoDB find() Statements
SELECT * FROM users	db.users.find()
SELECT id, user_id, status	db.users.find(

FROM users	{ }, { user_id: 1, status: 1 })
SELECT user_id, status FROM users	db.users.find({ }, { user_id: 1, status: 1, _id: 0 })
SELECT * FROM users WHERE status = "A"	db.users.find({ status: "A" })
SELECT user_id, status FROM users WHERE status = "A"	db.users.find({ status: "A" }, { user_id: 1, status: 1, _id: 0 })
SELECT * FROM users WHERE status != "A"	db.users.find({ status: { \$ne: "A" } })
SELECT * FROM users WHERE status = "A" AND age = 50	db.users.find({ status: "A", age: 50 })
SELECT * FROM users WHERE status = "A" OR age = 50	db.users.find({ \$or: [{ status: "A" } , { age: 50 }] })
SELECT * FROM users WHERE age > 25	db.users.find({ age: { \$gt: 25 } })
SELECT * FROM users WHERE age < 25	db.users.find({ age: { \$lt: 25 } })
SELECT * FROM users WHERE age < 25 AND age <= 50	db.users.find({ age: { \$gt: 25, \$lte: 50 } })
SELECT * FROM users WHERE user_id like "%bc%"	db.users.find({ user_id: /bc/ })

Table 7 SQL statements related to updating existing records in tables and the corresponding MongoDB statements

SQL Update Statements	MongoDB update() Statements
UPDATE users SET status = "C" WHERE age > 2	db.users.update({ age: { \$gt: 25 } }, { \$set: { status: "C" } }, { multi: true })
UPDATE users SET age = age + 3 WHERE status = "A"	db.users.update({ status: "A" }, { \$inc: { age: 3 } }, { multi: true })

Table 8 SQL statements related to deleting records from tables and the corresponding MongoDB statements

SQL Delete Statements	MongoDB remove() Statements
DELETE FROM users WHERE status = "D"	db.users.remove({ status: "D" })
DELETE FROM users	db.users.remove({ })

3.3. MongoDB Aggregation

Aggregations operations process data records and return computed results. Aggregation operations group values from multiple documents together, and can perform a variety of operations on the grouped data to return a single result. MongoDB provides a rich set of aggregation operations that examine and perform calculations on the data sets. Like queries, aggregation operations in MongoDB use collections of documents as an input and return results in the form of one or more documents (MongoDB manual).

The aggregation pipeline allows MongoDB to provide native aggregation capabilities that corresponds to many common data aggregation operations in SQL. Table 9 provides an overview of common SQL aggregation terms, functions, and concepts and the corresponding MongoDB aggregation operators (MongoDB manual).

Table 9 SQL aggregation terms, functions, and concepts and the corresponding MongoDB aggregation operators

SQL Terms, Functions, and Concepts	MongoDB Aggregation Operators
WHERE	\$match

GROUP BY	\$group
HAVING	\$match
SELECT	\$project
ORDER BY	\$sort
LIMIT	\$limit
SUM()	\$sum
COUNT()	\$sum
join	No direct corresponding operator; however, the \$unwind operator allows for somewhat similar functionality, but with fields embedded within the document.

Table 10 presents a quick reference of SQL aggregation statements and the corresponding MongoDB statements. The examples in the table assume the following conditions:

- The SQL examples assume two tables, orders and order_lineitem that join by the order_lineitem.order_id and the orders.id columns.
- The MongoDB examples assume one collection orders that contain documents of the following prototype:

```
{
  cust_id: "abc123",
  ord_date: ISODate("2012-11-02T17:04:11.102Z"),
  status: 'A',
  price: 50,
  items: [ { sku: "xxx", qty: 25, price: 1 },
           { sku: "yyy", qty: 25, price: 1 } ]
}
```

Table 10 SQL aggregation statements and the corresponding MongoDB statements

SQL Example	MongoDB Example	Description
SELECT COUNT(*) AS count FROM orders	db.orders.aggregate([{ \$group: { _id: null, count: { \$sum: 1 } }]	Count all records from orders

<pre>SELECT SUM(price) AS total FROM orders</pre>	<pre>} }]) db.orders.aggregate([{ \$group: { _id: null, total: { \$sum: "\$price" } } }])</pre>	<p>Sum the price field from orders</p>
<pre>SELECT cust_id, SUM(price) AS total FROM orders GROUP BY cust_id</pre>	<pre>db.orders.aggregate([{ \$group: { _id: "\$cust_id", total: { \$sum: "\$price" } } }])</pre>	<p>For each unique cust_id, sum the price field.</p>
<pre>SELECT cust_id, SUM(price) AS total FROM orders GROUP BY cust_id ORDER BY total</pre>	<pre>db.orders.aggregate([{ \$group: { _id: "\$cust_id", total: { \$sum: "\$price" } } }, { \$sort: { total: 1 } }])</pre>	<p>For each unique cust_id, sum the price field, results sorted by sum.</p>
<pre>SELECT cust_id, ord_date, SUM(price) AS total FROM orders GROUP BY cust_id, ord_date</pre>	<pre>db.orders.aggregate([{ \$group: { _id: { cust_id: "\$cust_id", ord_date: { month: { \$month: "\$ord_date" }, day: { \$dayOfMonth: "\$ord_date" }, year: { \$year: "\$ord_date" } } } } },</pre>	<p>For each unique cust_id, ord_date grouping, sum the price field. Excludes the time portion of the date.</p>

	total: { \$sum: "\$price" } } }])	
--	---	--

MongoDB aggregation operations are particularly handy in web applications such as online shopping portals where user features such as purchased items, scanned items, number of items, price of each item and total amount of purchased items have significant important for marketing purposes. In such applications, every action of every user can be bulk stored in the database and then by performing aggregation operations, shopping behavior/pattern of each user can be obtained. In chemical databases, these features are not of interest; data storage characteristics read and write operations are the major focus rather than aggregation.

3.4. MongoDB Indexing

Indexes support the efficient execution of queries in MongoDB. Without indexes, MongoDB must scan every document in a collection to select those documents that match the query statement. These collection scans are inefficient because they require mongod to process a larger volume of data than an index for each operation. Fundamentally, indexes in MongoDB are similar to indexes in other database systems such as RDBMS (MongoDB manual).

MongoDB defines indexes at the collection level and supports indexes on any field or sub-field of the documents in a MongoDB collection.

Limitations regarding indexing in MongoDB (Vanroose et. Thillo, 2014):

- collections : max 64 indexes
- index key length max 1024 bytes
- queries can only use 1 index [careful with concatenated indexes, negation, careful and regexp]
- indexes have storage requirements, and impact the performance of writes
- in memory sort (no-index) limited to 32 MB

4. MongoChem

Chemical databases are designed to store chemical information which is very diverse and comprehensive. They are mostly available in relational data format enables the users to perform database operations by SQL. Therefore, the interest is to have a different approach (NoSQL) when performing same/similar operations to investigate the differences.

4.1. Data Exploration

Within the scope of this study, CRUD operations are applied to a chemical database developed within a open chemistry project called MongoChem which uses MongoDB for data storage. First, MongoDB (shell version 2.4.9) is downloaded on the local host. By default, it is connected to a test database. Existing databases are shown with corresponding sizes. Then, the sample database which contains chemical information with some descriptors, images etc. is downloaded and connected to. The database has two collections: molecules and system.indexes; molecules collections will be used in the example. All the steps can be observed from the script below.

```
cansu@cansu:~$ mongo
MongoDB shell version: 2.4.9
connecting to: test
> show dbs
bookmarks  0.203125GB
chem       0.203125GB
local      0.078125GB
> use chem
switched to db chem
> show collections
molecules
system.indexes
```

The data is stored in a format similar to Chemical JSON which is intended to be expressive, and allow for most properties to be optional and encoded as arrays where appropriate. Chemical compound 'phenol' is shown below as an example to illustrate the data format (MongoChem).

```
{
  "_id" : ObjectId("4f54d9a3f0f6367ab7493f2d")
  "inchikey" : "ISWSIDIOOJBQZ-UHFFFAOYSA-N"
  "name" : "phenol"
  "bonds" : {
    "connections" : {
      "index" : [ 0, 1, 0, ...]
    },
    "order" : [ 1, 1, 2, 1, 1, 1, 2, 1, 2, 1, 1, 1, 1 ]
  },
  "descriptors" : {
    "tpsa" : 20.2,
    "xlogp3" : 0,
    "mass" : 94.11124,
    "vabc" : 89.95675835336127,
    "rotatable-bonds" : 0
  },
  "atoms" : {
    "elements" : {
```

```

        "number" : [ 8, 6, 6, ... ]
    },
    "coords" : {
        "3d" : [ -2.3622, 0.0001, ... ]
    },
},
"diagram" : BinData(0, "..."),
"fp2_fingerprint" : BinData(0, "..."),
"mass" : 94.11124,
"svg" : "<?xml version=\"1.0\"?>\n<svg vers ... >" ,
"inchi" : "InChI=1S/C6H6O/c7-6-4-2-1-3-5-6/h1-5,7H",
"atomCount" : 13,
"formula" : "C6H6O",
"heavyAtomCount" : 7
}

```

In the example above, a prototype of a document which is situated in 'molecules' collection of the chemical database is shown. The keys are written in bold and shown with their corresponding values. Embedded sub-documents can also be seen in the example; "descriptors", "bonds" and "atoms" have sub-documents at different levels.

Descriptions of each key are shown below to provide a better understanding of the content.

"name": Molecule name.

"atomCount": Total number of atoms in the molecule.

"heavyAtomCount": Total number of atoms but excluding the hydrogen atoms in the molecule.

"formula": Chemical formula.

"inchi": Chemical textual identifier (IUPAC International Chemical Identifier) which contains information about chemical formula, atom connections (prefix: c), hydrogen atoms (prefix: h). There can be more layers of informations such as charge and stereochemical layers contained in InChI identifier.

"inchikey": The condensed, 27 character standard InChIKey is a hashed version of the full standard InChI, designed to allow for easy web searches of chemical compounds (IUPAC, 2007).

"descriptors": All properties of molecules beyond their structure such as molecular mass and topological polar surface area (tpsa).

"diagram": It contains the binary data of png.

"svg": xml link of the svg image file of the molecule.

"bonds": It provides information about the atomic bonds of the molecule.

"fp2_fingerprint": Binary data file for 2D-fingerprint. Fingerprint is a widely used virtual screening concept in chemical similarity.

The data is indexed by "inchikey" (unique) and "heavyAtomCount" (non-unique) (MongoChem). All MongoDB collections have an index on the _id field that exists by default. If applications do not specify a value for _id the driver or the mongod will create an _id field with an ObjectId value. The _id index is unique, and prevents clients from inserting two documents with the same value for the _id field (MongoDB Manual).

4.2. Data Query and Modification

MongoDB queries return all fields in all matching documents in a collection by default. Projections can be included in the queries to limit the amount of data that MongoDB returns. The projections of results with a subset of fields enables the applications reduce their network overhead and processing requirements. Projections are the second argument to the find() method. They may either specify a list of fields to return or list fields to exclude in the result documents. Except for excluding the _id field in inclusive projections, mix exclusive and inclusive projections cannot be mixed (mongoDB manual).

```
> db.molecules.find({}, { name: 1 } )
```

```
{ "_id" : ObjectId("4f54d9a2f0f6367ab7493b84"), "name" : "(2-acetoxy-3-carboxy-propyl)-  
trimethyl-ammonium" }
```

```
{ "_id" : ObjectId("4f54d9a2f0f6367ab7493b85"), "name" : "5,6-dihydroxycyclohexa-1,3-diene-1-  
carboxylic acid" }
```

```
{ "_id" : ObjectId("4f54d9a2f0f6367ab7493b86"), "name" : "1-aminopropan-2-ol" }
```

```
{ "_id" : ObjectId("4f54d9a2f0f6367ab7493b87"), "name" : "(3-amino-2-keto-propyl) dihydrogen  
phosphate" }
```

```
{ "_id" : ObjectId("4f54d9a2f0f6367ab7493b88"), "name" : "1-chloro-2,4-dinitro-benzene" }
```

```
{ "_id" : ObjectId("4f54d9a2f0f6367ab7493b89"), "name" : "(9-ethylpurin-6-yl)amine" }
```

```
{ "_id" : ObjectId("4f54d9a2f0f6367ab7493b8a"), "name" : "2,3-dihydroxy-3-methyl-valeric  
acid" }
```

```
{ "_id" : ObjectId("4f54d9a2f0f6367ab7493b8b"), "name" : "(2,3,4,5,6-pentahydroxycyclohexyl)  
dihydrogen phosphate" }
```

```
{ "_id" : ObjectId("4f54d9a2f0f6367ab7493b8c"), "name" : "2-[[4-[(2-amino-4-keto-5,6,7,8-  
tetrahydro-1H-pteridin-6-yl)methyl-formyl-amino]benzoyl]amino]glutaric acid" }
```

```
{ "_id" : ObjectId("4f54d9a2f0f6367ab7493b8d"), "name" : "1,2-dichloroethane" }
```

```

{ "_id" : ObjectId("4f54d9a2f0f6367ab7493b8e"), "name" : "benzene-1,2,3,5-tetrol" }

{ "_id" : ObjectId("4f54d9a2f0f6367ab7493b8f"), "name" : "1,2,4-trichlorobenzene" }

{ "_id" : ObjectId("4f54d9a2f0f6367ab7493b90"), "name" : "7-[3-hydroxy-5-keto-2-(3-ketooct-1-enyl)cyclopentyl]enanthic acid" }

{ "_id" : ObjectId("4f54d9a2f0f6367ab7493b91"), "name" : "17-hydroxy-10,13-dimethyl-1,2,4,5,6,7,8,9,11,12,14,15,16,17-tetradecahydrocyclopenta[a]phenanthren-3-one" }

{ "_id" : ObjectId("4f54d9a2f0f6367ab7493b92"), "name" : "1,8-diazacyclotetradecane-2,9-quinone" }

{ "_id" : ObjectId("4f54d9a2f0f6367ab7493b93"), "name" : "2,3-dihydropyridine-2,6-dicarboxylic acid" }

{ "_id" : ObjectId("4f54d9a2f0f6367ab7493b94"), "name" : "4,5,6-trihydroxy-2,3-diketo-hexanoic acid" }

{ "_id" : ObjectId("4f54d9a2f0f6367ab7493b95"), "name" : "2,3-dihydroxybenzoic acid" }

{ "_id" : ObjectId("4f54d9a2f0f6367ab7493b96"), "name" : "3-(2,3-dihydroxyphenyl)propionic acid" }

{ "_id" : ObjectId("4f54d9a2f0f6367ab7493b97"), "name" : "2-ethyl-2-hydroxy-3-keto-butyric acid" }

```

Type "it" for more

As can be seen above, the query selects documents from molecules collection. It prints first 20 documents with corresponding fields; "name"s which was specified in the query (i.e. name: 1) and unique "id"s which is always in the projection by default unless otherwise is specified. There is no specific order in the returned documents if not specified, and rest of the documents can be printed by typing "it" in the command window.

In the query below, a selection operator is used to define the conditions for matching documents. It selects the documents that have "atomCount" greater than (i.e. \$gt) 5 and less than 10 (i.e. \$lt). This combination of operators can also be used to specify ranges. For performance enhancement, the size of the result can be constrained by limiting the amount data to be received by the application over the network. The limit () method which specifies the number of documents as 5 (i.e. limit(5)) in the result is applied on the cursor. Then the sort() modifier sorts the results by name in ascending order.

```
> db.molecules.find( { atomCount: { $gt: 5, $lte: 10 } }, { name:1, atomCount:1, _id: 0 } ).limit(5).sort( {name: 1 } )
```

```
{ "atomCount" : 8 }
```

```
{ "atomCount" : 6 }
```

```
{ "name" : "", "atomCount" : 6 }
```

```
{ "atomCount" : 9, "name" : "(methylthio)methane" }
```

```
{ "atomCount" : 8, "name" : "1,2-dichloroethane" }
```

As can be seen above, 5 documents whose "atomCount" field has a value in range of 5 to 10 sorted by their name are printed in the result. Only the "atomCount" and name fields are printed due to the projection criteria set as before. This indicates that MongoDB narrows the query by scanning the range of documents defined with query criteria. Thus, the query selects the documents using an index which results in a more efficient scanning, consequently lower query time.

In the query results, it can easily be observed that two of the documents do not have a name field and one does not have a value for the name field. This can be interpreted as existence of dummy data in this database.

Another important point is the placement of the fields which is directly related with the document-based database characteristics. One can observe that the name field is placed before than "atomCount" field in one of the documents in the result as opposed to the other documents. It indicates how that specific document (data) was inserted to the database i.e. the order of fields. In a relational database, it wouldn't matter as the fields and their values are set in the data model regardless of the insertion order.

MongoDB enables data storage in sub-documents as well which is an important feature. Level of hierarchy with respect to its field has to be specified when querying sub-documents. It provides information about the relations between fields which is not possible in relational databases, since every field has to be represented with a separate column in the table. If the array index of the embedded document is known, the document can be specified by using the sub-document's position using the dot notation. Otherwise, the name of the field that contains the array can be concatenated with a dot (.) and the name of the field in the sub-document. An illustration of dot notation can be seen below.

```
> db.molecules.find( { "descriptors.tpsa" : 20.2, "descriptors.xlogp3" : 0 }, { name: 1, descriptors: 1, _id: 0 } )
```

```
{ "name" : "2-chloroethanol", "descriptors" : { "tpsa" : 20.2, "new boiling_point" : 127, "new boiling point" : 127, "vabc" : 67.14972242352783, "boiling_point" : 127, "new melting_point" : -63, "boiling" : 127, "mass" : 80.5135, "melting_point" : -63, "xlogp3" : 0, "rotatable-bonds" : 1, "melting jlk" : -63 } }
```

```
{ "descriptors" : { "tpsa" : 20.2, "xlogp3" : 0, "mass" : 639.0474, "rotatable-bonds" : 31, "vabc" : 760.7044737816262 }, "name" : "2-(3,7,11,15,19,23,27,31-octamethyldotriaconta-2,6,10,14,18,22,26,30-octaenyl)phenol" }
```

```
{ "name" : "2,5-dichlorophenol", "descriptors" : { "tpsa" : 20.2, "xlogp3" : 0, "mass" : 163.00136, "vabc" : 120.3788938942279, "rotatable-bonds" : 0 } }
```

```
{ "name" : "phenylmethanol", "descriptors" : { "tpsa" : 20.2, "new boiling_point" : 205, "new
```

boiling_point" : 205, "vabc" : 107.2527429792279, "boiling_point" : 205, "new melting_point" : -15, "boiling" : 205, "mass" : 108.13782, "melting_point" : -15, "xlogp3" : 0, "rotatable-bonds" : 1, "melting_jlk" : -15 } }

{ "name" : "butan-1-ol", "descriptors" : { "tpsa" : 20.2, "xlogp3" : 0, "mass" : 74.1216, "vabc" : 86.53062390482788, "rotatable-bonds" : 2 } }

{ "name" : "17-(1,5-dimethylhexyl)-10,13-dimethyl-2,3,4,7,8,9,11,12,14,15,16,17-dodecahydro-1H-cyclopenta[a]phenanthren-3-ol", "descriptors" : { "tpsa" : 20.2, "xlogp3" : 0, "mass" : 386.65354, "vabc" : 432.2759767957644, "rotatable-bonds" : 5 } }

{ "name" : "3-phenylprop-2-en-1-ol", "descriptors" : { "tpsa" : 20.2, "xlogp3" : 0, "mass" : 134.1751, "vabc" : 139.2082535301613, "rotatable-bonds" : 3 } }

{ "name" : "p-cumenylmethanol", "descriptors" : { "tpsa" : 20.2, "xlogp3" : 0, "mass" : 150.21756, "vabc" : 159.140696856828, "rotatable-bonds" : 2 } }

{ "name" : "o-cresol", "descriptors" : { "tpsa" : 20.2, "xlogp3" : 0, "mass" : 108.13782, "vabc" : 107.2527429792279, "rotatable-bonds" : 0 } }

{ "name" : "m-cresol", "descriptors" : { "tpsa" : 20.2, "xlogp3" : 0, "mass" : 108.13782, "vabc" : 107.2527429792279, "rotatable-bonds" : 0 } }

{ "name" : "octan-1-ol", "descriptors" : { "tpsa" : 20.2, "xlogp3" : 0, "mass" : 130.22792, "vabc" : 155.7145624082945, "rotatable-bonds" : 6 } }

{ "name" : "2,3,4,5,6-pentachlorophenol", "descriptors" : { "tpsa" : 20.2, "xlogp3" : 0, "mass" : 266.33654, "vabc" : 166.0120972055278, "rotatable-bonds" : 0 } }

{ "name" : "phenol", "descriptors" : { "tpsa" : 20.2, "xlogp3" : 0, "mass" : 94.11124, "vabc" : 89.95675835336127, "rotatable-bonds" : 0 } }

{ "name" : "propan-1-ol", "descriptors" : { "tpsa" : 20.2, "xlogp3" : 0, "mass" : 60.09502, "vabc" : 69.23463927896123, "rotatable-bonds" : 1 } }

{ "name" : "3,7-dimethyl-9-(2,6,6-trimethylcyclohexen-1-yl)nona-2,4,6,8-tetraen-1-ol", "descriptors" : { "tpsa" : 20.2, "xlogp3" : 0, "mass" : 286.4516, "vabc" : 337.7276257138959, "rotatable-bonds" : 9 } }

{ "name" : "2,4,6-tribromophenol", "descriptors" : { "tpsa" : 20.2, "xlogp3" : 0, "mass" : 330.79942, "vabc" : 147.8076152860614, "rotatable-bonds" : 0 } }

{ "name" : "4-chloro-3-methyl-phenol", "descriptors" : { "tpsa" : 20.2, "xlogp3" : 0, "mass" : 142.58288, "vabc" : 122.4638107496612, "rotatable-bonds" : 0 } }

{ "name" : "4-nonylphenol", "descriptors" : { "tpsa" : 20.2, "xlogp3" : 0, "mass" : 220.35046, "vabc" : 245.6206199861614, "rotatable-bonds" : 8 } }

As can be seen above, the field "descriptors" has several embedded sub-documents (i.e. tpsa, xlogp3, mass, vabc, rotatable-bonds) and the query has selection criteria regarding two of them (i.e. "descriptors.tpsa" : 20.2, "descriptors.xlogp3" : 0). It projects the name and descriptors of the corresponding molecules.

In MongoDB, sub-documents can have sub-documents which indicates higher level of hierarchy of the fields. When the field involves an array, a specific array or for specific values in an array can be queried. If the array involves embedded documents, specific fields in the embedded documents can also be queried by using dot notation. For fields that contain arrays, MongoDB provides the following projection operators: \$elemMatch, \$slice and \$. Explicit storage of arrays in a relational database table (SQL) is not possible; however, there are some methods for storing arrays by converting formats. An example of explicit storing of arrays in MongoDB can be seen below.

```
> db.molecules.findOne( { "atoms.elements.number" : { $gt: 5 } }, {name: 1, _id: 0, atoms: 1 } )

{ "atoms" : { "elements" : { "number" : [ 8,      8,      8,      8,      7, 6,  6,      6,      6,
                                         6, 6,   6,   6,   6,   1,   1, 1,  1,   1,
                                         1, 1,   1,   1,   1,   1,   1,   1, 1,  1,
                                         1, 1,   1 ] }, "coords" : { "3d" : [ -0.1902,
1.3409, 0.3005,      2.448, -2.3678,      1.071,
2.2391, -1.9287,      -1.1496,      1.2775,
2.3629, -1.1618,      -2.1546,      -0.6965,      -
0.0209, -0.7847, -0.9897,      0.4738,      0.1977,
0.0264, -0.1011,      -2.2003,      -0.7415,      -
1.5539, -3.1585, -1.7242,      0.5181,      -2.6089,
0.698,  0.4303,      1.6026,      -0.2265,
0.4447, 2.1184,      -1.5795,      0.0175,
0.6202, 2.3512,      -0.1299,      0.5936,
3.4751, 0.8612,      -0.8149,      -0.9382,
1.5693, -0.5333,      -2.0143,      0.177, 0.2374,
-0.0302, -1.1943,      -1.6769,      0.1203,      -
1.9745, -1.7507, -1.6822,      -1.8849,      -3.25, -0.6979,
-1.8617, -3.155,      -1.6611,      1.6102,      -
4.1514, -1.493, 0.1213,      -2.838,      -2.717,
0.1888, -2.2788,      0.8768,      1.458, -2.2939,
1.4619, -0.2844,      -3.7052,      0.7061,
0.4169, 1.598, -0.16, 1.5404,      2.3437,
0.507,  0.1121,      0.9681,      3.1259,
1.8267, -0.4251,      3.8589,      0.9587,
1.2389, 4.2848,      0.5088,      2.7859,      -
3.2422, 0.7823 ] } }, "name" : "(2-acetoxy-3-carboxy-propyl)-
trimethyl-ammonium" }
```

As can be seen above, `findOne()` method which returns a single document from a collection is used. The `findOne()` method takes the same parameters as `find()`, but returns a single document rather than a cursor (i.e. 20 documents by default in MongoDB). The field "atoms" has a sub-document with the field "elements" which has a sub-document with the field "number". The query prints a document which satisfies a condition (i.e. $\$gt: 5$) in the "number" field. The array in the corresponding sub-document must contain at least one element that satisfies the conditions. Explicit storage of arrays are useful and more human readable when analyzing or updating/changing the molecule structure data stored in an array.

5. Pymatgen

Pymatgen (Python Materials Genomics) is a robust, open-source Python library for materials analysis. Pymatgen currently powers the public Materials Project which is an initiative to make calculated properties of all known inorganic materials available to materials researchers.

A key enabler in high-throughput computational materials science efforts is a robust set of software tools to perform initial setup for calculations (e.g. generation of structures and necessary input files) and post-calculation analysis to derive useful material properties from raw calculated data. The aims of pymatgen are as follows (Ong et al., 2013):

- Define core Python objects for materials data representation.
- Provide a well-tested set of structure and thermodynamic analysis tools relevant to many applications.
- Establish an open platform for researchers to collaboratively develop sophisticated analyses of materials data obtained both from first principles calculations and experiments.

Pymatgen is examined with illustration of some examples, since its infrastructure uses MongoDB for materials properties as a central component. It serves several roles:

- Workflow manager for managing the state of high-throughput calculations (performed by high-throughput workflow engine developed within Materials Project).
- Execution engine for storage and analytics for the calculation results.
- Back-end repository as a searchable back-end for data dissemination.

Flexibility and scalability MongoDB is leveraged to achieve these three goals within the same deployment. Unlike RDBMS products such as PostgreSQL, it does not require a schema between all the data types which involves outputs and views of the calculated materials properties, execution state and outputs. The data being stored is continually evolving as new types of calculations and collaborators are added on to the project over time (Gunter et. al., 2012).

MongoDB has a powerful and simple query language named as Mongo Query Language. Ease of administration and good performance on read-heavy workloads where most of the data can fit into

memory are other advantages of MongoDB. The organization of the data store to serve multiple overlapping roles is described as follows (Gunter et. al., 2012):

- **Input data:** The input data is the standard JSON representation of a crystal and its metadata which may come from a user or an external data source. Essential information that must be stored and accessed is standard physical characteristics (atomic masses, positions, etc.), and metadata indicating the source of the crystal.
- **Execution state data:** The datastore is also used as a task queue, and so must be able to represent the state and intermediate results for all tasks in the system. The workflow engine needs to insert and remove tasks from this queue. Errors and selected results are necessary for the logic of restarting or modifying workflows. The connection between runnable tasks and their results needs to be preserved at all times. The representations used for the task data must adapt to changes in both the workflow engine and the result data.

All the execution state are stored in two database collections: engines and tasks. The engines collection contains jobs that are waiting to be run, running, and completed. The jobs are modeled as black boxes of inputs, including only those outputs needed for control logic. Jobs can be selected using MongoDB queries on the inputs which provides mechanism for matching types of jobs to types of resources.

- **Calculated property data:** There are several types of calculated properties that must be stored which includes materials, phase diagrams, x-ray diffraction patterns, and bandstructures. There needs to be a connection between the calculated properties, the execution that produced them, and the input data. Each type of calculated properties is given its own collection in the database. These include phase diagrams and diffraction patterns. New properties can be added as new collections. The canonical class of properties is stored in the materials collection which is a view of the properties needed by the Web API and Web UI for each material.

MongoDB provides flexibility to add new types of calculations and properties. A simplification is held for number of APIs (application programming interface), servers, and ways of querying data by storing all semi-structured and structured data types in MongoDB. Though the database is schemaless, programmatic changes in the data layout can still perturb other components. To mitigate this, an abstraction layer is implemented for queries and updates to the main collections. This layer allows us to install convenient aliases for deeply nested fields or change the names of collections in a single central place. The intermediate layer also provides a defense against lock-in to MongoDB's query language: the QueryEngine could completely transform input queries and updates for another datastore (Gunter et. al., 2012).

5.1. Pymatgen Examples

To illustrate the power of the pymatgen library, a few examples are presented. Scripts for all the examples can be found in the Appendix.

Pymatgen contains a set of core classes to represent an Element, Specie and Composition. These objects contains useful properties such as atomic mass and ionic radii. These core classes are loaded by default with pymatgen. For Silicon, these properties can be printed as follows:

Atomic mass of Si is 28.0855 amu

Si has a melting point of 1687 K

Ionic radii for Si: {4: 0.54}

The units are printed for atomic masses and ionic radii. Pymatgen comes with a complete system of managing units in pymatgen.core.unit.

A Composition is essentially an immutable mapping of Elements/Species with amounts, and useful properties such as molecular weight and get_atomic_fraction. Either an Element/Specie object or a string can be used as keys. For Fe₂O₃ molecule, weight, amount, atomic fraction and weight fraction of Fe atoms in the molecule can be printed as follows:

Weight of Fe₂O₃ is 159.6882 amu

Amount of Fe in Fe₂O₃ is 2.0

Atomic fraction of Fe is 0.4

Weight fraction of Fe is 0.699425505454

A Lattice represents a Bravais lattice. Convenience static functions are provided for the creation of common lattice types from a minimum number of arguments such as lattice lengths and angles. A Structure object represents a crystal structure (lattice and basis). A Structure is essentially a list of PeriodicSites with the same Lattice. Pymatgen also provides many analyses functions for Structures. Creation of a structure object for CsCl molecule is done, and unit cell volume and first site of the structure are printed as follows:

Unit cell vol = 74.088

First site of the structure is [0. 0. 0.] Cs

Pymatgen also provides IO support for various file formats in the pymatgen.io package. A convenient set of read_structure and write_structure functions are also provided which auto-detects several well-known formats such as POSCAR (contains information about the position of the ions).

Electronic structure which is the state of motion of electrons in an electrostatic field created by stationary nuclei is drawn for Fe atom as illustration of a property. As mentioned before, full electronic structure information is available in Pytmatgen library by default. Matplotlib which is a Python 2D plotting library is used for drawing of the electron structure of Fe atom. It can be seen in Figure 6.

Electronic structure of Fe

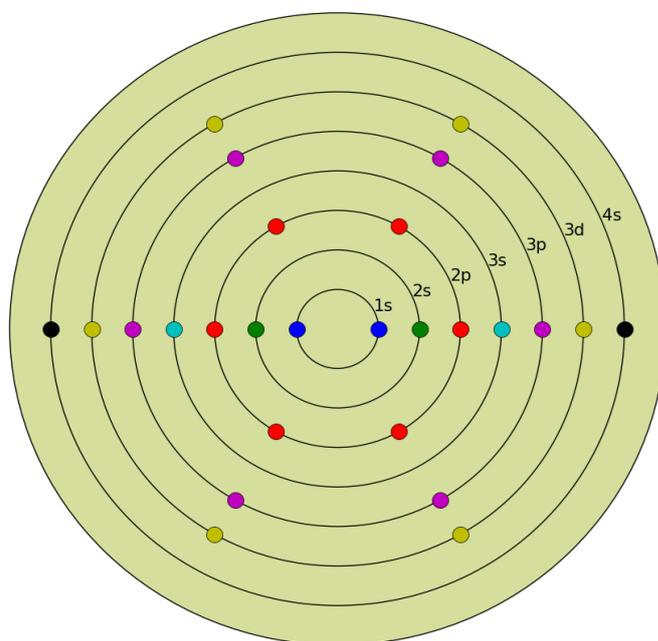


Figure 6 Electronic structure of Fe atom

6. Discussion and Conclusions

Choice of the database model is strictly related to the application itself. In this study, MongoDB is examined for chemical data storage and retrieval purposes. Chemical databases have historically been developed in relational model and the operations have been held by SQL. Due to increasing complexity and amount of semi-structured data, MongoDB has gained popularity for chemical data storage.

Flexible and schemaless data model of MongoDB provides the opportunity for storage of any type of data which can be inserted by several different users without imposing a certain data structure. It also enables the users to create and handle very complex and sophisticated data model which is usually the case for chemical data, especially for chemical structure information for complex molecules. In SQL databases, a table has to be created/determined and declared before inserting data which might result in longer query times and complexity in coding. Sub-document creation, insertion and storage are also possible at different levels of hierarchy without any limitation. It is a beneficial feature as chemical data types, contents and formats are variant. In SQL databases, sub-document have to be defined and stored in columns which requires relatively complex coding, and prevent the user to see the hierarchical relations between the fields. Explicit array storage in MongoDB has a great importance in case of chemical structure data storage which is not directly possible in SQL databases.

Despite the bulk storage of huge amount of data, MongoDB does not suffer from long processing times due to horizontal scaling ability. Once the data amount increases, more machines can be add to the cluster to enhance processing performance and prevent associated downtime errors. Therefore, there is no limit for scaling while scaling (vertical) in SQL is limited with the hardware improvement of the machine. In case of development of a chemical database, it enables many users to work simultaneously without suffering from scaling issues. Thus, horizontal scaling is the most important advantage of NoSQL over SQL.

MongoDB is designed to favor availability over consistency (ACID vs. BASE). Depending on the application, sacrificing from consistency might not be an important trade-off. For chemical databases, this trade-off is not significant. However, it remains as one of the most important concerns in implementation of NoSQL.

Another obstacle in wider implementation of NoSQL in general is the immaturity. Maturity refers both to how old a particular system is and to how thoroughly tested it is. More tests means more time to ensure system stability, more bugs identified, and more questions answered. SQL databases have been around a lot longer time and used by far more users in wider variety of applications with higher user support in comparison with NoSQL databases. However, open-source nature of NoSQL might offset this drawback. Since it is open to public, any user can fix the bugs which also results in lower development costs. Free use of NoSQL is another advantage.

Security is an important aspect in DBMSs. Designers of MongoDB were primarily concerned about the security, thus there are some security concerns. MongoDB and other NoSQL databases are still very new in the field with respect to their feature set, especially with the respect to security, so fine-grained permissions or access control in these systems are yet to be provided. Based on this MongoDB or NoSQL databases in general lose in an adoptability study for large mission critical

enterprises. Often developers need to devise scopes to maximize its advantages and minimize its weaknesses (Shermin, 2013).

Both SQL and NoSQL approaches have limitations and advantages which strongly depend on the types of the application, its requirements and priorities. Therefore, a careful analysis has to be done when deciding on DBMS. Flaws of NoSQL approach are mainly due to the immaturity of the concept, thus further development would make it more desirable resulting in a greater deployment.

References

1. Codd, E. F. (1970). A relational model of data for large shared data banks. *Communications of the ACM*, 13(6), 377-387.
2. Fowler, M., & Sadalage, P. (2013). *NoSQL Distilled*. Addison Wesley.
3. Shermin, M. (2013). An Access Control Model for NoSQL Databases.
4. Scofield, Ben. (2010). NoSQL – Death to Relational Databases(?)– Presentation at the CodeMash conference in Sandusky (Ohio). Accessed October 5, 2014. url: <http://www.slideshare.net/bescofield/nosql-codemash-2010>.
5. Popescu, Alex. (2010). Presentation: An Interesting NoSQL categorization. Accessed October 13, 2014. url: <http://nosql.mypopescu.com/post/396337069/presentation-nosql-codemash-an-interestingnosql>.
6. Brewer, E. A. (2000, July). Towards robust distributed systems. In *PODC* (p. 7).
7. Pritchett, D. (2008). Base: An acid alternative. *Queue*, 6(3), 48-55.
8. Vanroose. Peter., Thillo V. Kris (2014). ACID or BASE? - the case of NoSQL. Accessed October 15, 2014. url: <http://www.abis.be/resources/presentations/gsebedb220140612nosql.pdf>.
9. Strauch, C. (2011). NoSQL Databases. Accessed November 15, 2014. url: <http://www.christof-strauch.de/nosql dbs.pdf>.
- 10 Stonebraker, M., Madden, S., Abadi, D. J., Harizopoulos, S., Hachem, N., & Helland, P. (2007, September). The end of an architectural era:(it's time for a complete rewrite). In *Proceedings of the 33rd international conference on Very large data bases* (pp. 1150-1160). VLDB Endowment.
11. MongoDB. MongoDB Documentation, Release 2.6.4. Accessed September 20, 2014. url: <http://docs.mongodb.org/master/MongoDB-manual.pdf>
12. Open Chemistry Project. Accessed October 17, 2014. url: <http://wiki.openchemistry.org/MongoChem>
13. "The IUPAC International Chemical Identifier (InChI)". IUPAC. 5 September 2007. Retrieved 2007-09-18.
14. Pymatgen. Accessed October 17, 2014. url: <http://pymatgen.org/>
15. Materials Project. Accessed October 17, 2014. url: <http://www.materialsproject.org>
16. Ong, S. P., Richards, W. D., Jain, A., Hautier, G., Kocher, M., Cholia, S., ... & Ceder, G. (2013). Python Materials Genomics (pymatgen): A robust, open-source python library for materials analysis. *Computational Materials Science*, 68, 314-319.
17. Gunter, D., Cholia, S., Jain, A., Kocher, M., Persson, K., Ramakrishnan, L., ... & Ceder, G.

(2012, November). Community Accessible Datastore of High-Throughput Calculations: Experiences from the Materials Project. In *High Performance Computing, Networking, Storage and Analysis (SCC), 2012 SC Companion:* (pp. 1244-1251). IEEE

18. Vicknair, C., Macias, M., Zhao, Z., Nan, X., Chen, Y., & Wilkins, D. (2010, April). A comparison of a graph database and a relational database: a data provenance perspective. In *Proceedings of the 48th annual Southeast regional conference* (p. 42). ACM.

19. Dede, E., Govindaraju, M., Gunter, D., Canon, R. S., & Ramakrishnan, L. (2013, June). Performance evaluation of a mongodb and hadoop platform for scientific data analysis. In *Proceedings of the 4th ACM workshop on Scientific cloud computing* (pp. 13-20). ACM.

Appendix

Python scripts for basic functionality examples of Pymatgen.

```
cansu@cansu:~$ python
```

```
Python 2.7.6 (default, Mar 22 2014, 22:59:56)
```

```
[GCC 4.8.2] on linux2
```

Type "help", "copyright", "credits" or "license" for more information.

```
>>> import pymatgen as mg
```

```
/home/cansu/.local/lib/python2.7/site-packages/setuptools-7.0-py2.7.egg/pkg_resources.py:1045:  
UserWarning: /home/cansu/.python-eggs is writable by group/others and vulnerable to attack when  
used with get_resource_filename. Consider a more secure location (set with .set_extraction_path or  
the PYTHON_EGG_CACHE environment variable).
```

```
>>> si = mg.Element("Si")
```

```
>>> print("Atomic mass of Si is {}".format(si.atomic_mass))
```

```
Atomic mass of Si is 28.0855 amu
```

```
>>> print("Si has a melting point of {}".format(si.melting_point))
```

```
Si has a melting point of 1687 K
```

```
>>> print("Ionic radii for Si: {}".format(si.ionic_radii))
```

```
Ionic radii for Si: {4: 0.54}
```

```
>>> print("Atomic mass of Si in kg: {}".format(si.atomic_mass.to("kg")))
```

```
Atomic mass of Si in kg: 4.66370658657e-26 kg
```

```
>>> print("Atomic mass of Si in kg: {}".format(si.atomic_mass.to("kg")))
```

```
Atomic mass of Si in kg: 4.66370658657e-26 kg
```

```
>>> fe2 = mg.Specie("Fe", 2)
```

```
>>> print(fe2.atomic_mass)
```

```
55.845 amu
```

```
>>> print(fe2.ionic_radius)
```

```
0.92 ang
```

```

>>> comp = mg.Composition("Fe2O3")

>>> print("Weight of Fe2O3 is {}".format(comp.weight))

Weight of Fe2O3 is 159.6882 amu

>>> print("Amount of Fe in Fe2O3 is {}".format(comp["Fe"]))

Amount of Fe in Fe2O3 is 2.0

>>> print("Atomic fraction of Fe is {}".format(comp.get_atomic_fraction("Fe")))

Atomic fraction of Fe is 0.4

>>> print("Weight fraction of Fe is {}".format(comp.get_wt_fraction("Fe")))

Weight fraction of Fe is 0.699425505454

>>> lattice = mg.Lattice.cubic(4.2)

>>> print(lattice.lengths_and_angles)

((4.2000000000000002, 4.2000000000000002, 4.2000000000000002), (90.0, 90.0, 90.0))

>>> structure = mg.Structure(lattice, ["Cs", "Cl"], [[0, 0, 0], [0.5, 0.5, 0.5]])

>>> print("Unit cell vol = {}".format(structure.volume))

Unit cell vol = 74.088

>>> print("First site of the structure is {}".format(structure[0]))

First site of the structure is [ 0.  0.  0.] Cs

>>> structure.make_supercell([2, 2, 1])

>>> del structure[0]

>>> structure.append("Na", [0,0,0])

>>> structure[-1] = "Li"

>>> structure[0] = "Cs", [0.01, 0.5, 0]

>>> immutable_structure = mg.IStructure.from_sites(structure)

>>> print(immutable_structure)

Structure Summary (Cs3 Li1 Cl4)

Reduced Formula: Cs3LiCl4

```

```

abc : 8.400000 8.400000 4.200000
angles: 90.000000 90.000000 90.000000
Sites (8)
1 Cs 0.010000 0.500000 0.000000
2 Cs 0.500000 0.000000 0.000000
3 Cs 0.500000 0.500000 0.000000
4 Cl 0.250000 0.250000 0.500000
5 Cl 0.250000 0.750000 0.500000
6 Cl 0.750000 0.250000 0.500000
7 Cl 0.750000 0.750000 0.500000
8 Li 0.000000 0.000000 0.000000

```

```
>>> structure.to(filename="POSCAR")
```

```
>>> print(structure.to(fmt="poscar"))
```

```
Cs3 Li1 Cl4
```

```
1.0
```

```
8.400000 0.000000 0.000000
```

```
0.000000 8.400000 0.000000
```

```
0.000000 0.000000 4.200000
```

```
Cs Cl Li
```

```
3 4 1
```

```
direct
```

```
0.010000 0.500000 0.000000 Cs
```

```
0.500000 0.000000 0.000000 Cs
```

```
0.500000 0.500000 0.000000 Cs
```

```
0.250000 0.250000 0.500000 Cl
```

```
0.250000 0.750000 0.500000 Cl
```

0.750000 0.250000 0.500000 Cl

0.750000 0.750000 0.500000 Cl

0.000000 0.000000 0.000000 Li

```
>>> structure.to(filename="CsCl.cif")
```

```
>>> print(structure.to(fmt="cif"))
```

```
#generated using pymatgen
```

```
data_Cs3LiCl4
```

```
_symmetry_space_group_name_H-M 'P 1'
```

```
_cell_length_a 8.40000000
```

```
_cell_length_b 8.40000000
```

```
_cell_length_c 4.20000000
```

```
_cell_angle_alpha 90.00000000
```

```
_cell_angle_beta 90.00000000
```

```
_cell_angle_gamma 90.00000000
```

```
_symmetry_Int_Tables_number 1
```

```
_chemical_formula_structural Cs3LiCl4
```

```
_chemical_formula_sum 'Cs3 Li1 Cl4'
```

```
_cell_volume 296.352
```

```
_cell_formula_units_Z 1
```

```
loop_
```

```
_symmetry_equiv_pos_site_id
```

```
_symmetry_equiv_pos_as_xyz
```

```
1 'x, y, z'
```

```
loop_
```

```
_atom_site_type_symbol
```

```

_atom_site_label
_atom_site_symmetry_multiplicity
_atom_site_fract_x
_atom_site_fract_y
_atom_site_fract_z
_atom_site_occupancy

Cs Cs1 1 0.010000 0.500000 0.000000 1
Cs Cs2 1 0.500000 0.000000 0.000000 1
Cs Cs3 1 0.500000 0.500000 0.000000 1
Cl Cl4 1 0.250000 0.250000 0.500000 1
Cl Cl5 1 0.250000 0.750000 0.500000 1
Cl Cl6 1 0.750000 0.250000 0.500000 1
Cl Cl7 1 0.750000 0.750000 0.500000 1
Li Li8 1 0.000000 0.000000 0.000000 1

```

```

>>> structure = mg.Structure.from_file("POSCAR")
>>> from pymatgen.io.vaspio_set import MPVaspInputSet
>>> v = MPVaspInputSet()
>>> v.write_input(structure, "MyInputFiles")

```

Traceback (most recent call last):

```

File "<stdin>", line 1, in <module>

File          "/home/cansu/.local/lib/python2.7/site-packages/pymatgen-3.0.7-py2.7-linux-
x86_64.egg/pymatgen/io/vaspio_set.py", line 151, in write_input

    for k, v in self.get_all_vasp_input(structure).items():

File          "/home/cansu/.local/lib/python2.7/site-packages/pymatgen-3.0.7-py2.7-linux-
x86_64.egg/pymatgen/io/vaspio_set.py", line 132, in get_all_vasp_input

    d['POTCAR'] = self.get_potcar(structure)

```

File `"/home/cansu/.local/lib/python2.7/site-packages/pymatgen-3.0.7-py2.7-linux-x86_64.egg/pymatgen/io/vaspio_set.py"`, line 310, in `get_potcar`

```
return Potcar(self.get_potcar_symbols(structure))
```

File `"/home/cansu/.local/lib/python2.7/site-packages/pymatgen-3.0.7-py2.7-linux-x86_64.egg/pymatgen/io/vaspio/vasp_input.py"`, line 1315, in `__init__`

```
self.set_symbols(symbols, functional, sym_potcar_map)
```

File `"/home/cansu/.local/lib/python2.7/site-packages/pymatgen-3.0.7-py2.7-linux-x86_64.egg/pymatgen/io/vaspio/vasp_input.py"`, line 1390, in `set_symbols`

```
p = PotcarSingle.from_symbol_and_functional(el, functional)
```

File `"/home/cansu/.local/lib/python2.7/site-packages/pymatgen-3.0.7-py2.7-linux-x86_64.egg/pymatgen/io/vaspio/vasp_input.py"`, line 1238, in `from_symbol_and_functional`

```
"POTCAR.{}".format(symbol)),
```

File `"/usr/lib/python2.7/posixpath.py"`, line 77, in `join`

```
elif path == " or path.endswith('/'):
```

AttributeError: 'NoneType' object has no attribute 'endswith'

Python script for electron structure example of Pymatgen.

```
cansu@cansu:~$ sudo apt-get install python-matplotlib-doc python-scipy python-traits texlive-extra-utils
```

```
>>> from pymatgen import Element
```

```
/home/cansu/.local/lib/python2.7/site-packages/setuptools-7.0-py2.7.egg/pkg_resources.py:1045: UserWarning: /home/cansu/.python-eggs is writable by group/others and vulnerable to attack when used with get_resource_filename. Consider a more secure location (set with .set_extraction_path or the PYTHON_EGG_CACHE environment variable).
```

```
>>> symbol = "Fe"
```

```
>>> el = Element(symbol)
```

```
>>> elec_structure = el.full_electronic_structure
```

```
>>> matplotlib inline
```

File "`<stdin>`", line 1

```
matplotlib inline
```

```
^
```

```
SyntaxError: invalid syntax
```

```
>>> from matplotlib.pyplot import figure, show, rc, grid
```

```
>>> rc('grid', color='#316931', linewidth=0, linestyle='-')
```

```
>>> rc('xtick', labelsizes=15)
```

```
>>> rc('ytick', labelsizes=15)
```

```
>>> import numpy as np
```

```
>>> fig = figure(figsize=(10, 10))
```

```
>>> ax = fig.add_axes([0.1, 0.1, 0.8, 0.8], polar=True, axisbg='#d5de9c')
```

```
>>> rticks = []
```

```
>>> rlabels = []
```

```
>>> for i, shell in enumerate(elec_structure):
```

```
... r = 0.5 * (i + 1)
```

```
File "<stdin>", line 2
```

```
    r = 0.5 * (i + 1)
```

```
^
```

```
IndentationError: expected an indented block
```

```
>>> rad = [r] * 1000
```

```
File "<stdin>", line 1
```

```
    rad = [r] * 1000
```

```
^
```

```
IndentationError: unexpected indent
```

```
>>> theta = [2 * np.pi * j / 1000 for j in xrange(1000)]
```

```
File "<stdin>", line 1
```

```
    theta = [2 * np.pi * j / 1000 for j in xrange(1000)]
```

^

IndentationError: unexpected indent

```
>>> ax.plot(theta, rad, 'k-', lw=1)
```

File "<stdin>", line 1

```
ax.plot(theta, rad, 'k-', lw=1)
```

^

IndentationError: unexpected indent

```
>>> for i, shell in enumerate(elec_structure):
```

```
... r = 0.5 * (i + 1)
```

File "<stdin>", line 2

```
r = 0.5 * (i + 1)
```

^

IndentationError: expected an indented block

```
>>> for i, shell in enumerate(elec_structure):
```

```
... r = 0.5 * (i + 1)
```

```
... rad = [r] * 1000
```

```
... theta = [2 * np.pi * j / 1000 for j in xrange(1000)]
```

```
... ax.plot(theta, rad, 'k-', lw=1)
```

```
... rad = [r] * shell[2]
```

```
... theta = [2 * np.pi * j / shell[2] for j in xrange(shell[2])]
```

```
... ax.plot(theta, rad, 'o', markersize=15)
```

```
... rticks.append(r)
```

```
... rlabels.append("{} {}".format(shell[0], shell[1]))
```

```
... ax.set_rmax(r + 0.5)
```

File "<stdin>", line 11

```
ax.set_rmax(r + 0.5)
```

^

SyntaxError: invalid syntax

```
>>> ax.set_rmax(r + 0.5)
```

File "<stdin>", line 1

```
ax.set_rmax(r + 0.5)
```

^

IndentationError: unexpected indent

```
>>> for i, shell in enumerate(elec_structure):
```

```
... #Draw the shell
```

```
... r = 0.5 * (i + 1)
```

```
... rad = [r] * 1000
```

```
... theta = [2 * np.pi * j / 1000 for j in xrange(1000)]
```

```
... ax.plot(theta, rad, 'k-', lw=1)
```

```
... #Draw the electrons
```

```
... rad = [r] * shell[2]
```

```
... theta = [2 * np.pi * j / shell[2] for j in xrange(shell[2])]
```

```
... ax.plot(theta, rad, 'o', markersize=15)
```

```
... rticks.append(r)
```

```
... rlabels.append("{} {}".format(shell[0], shell[1]))
```

```
... ax.set_rmax(r + 0.5)
```

File "<stdin>", line 13

```
ax.set_rmax(r + 0.5)
```

^

IndentationError: unexpected indent

```
>>> ax.set_rmax(r + 0.5)
```

File "<stdin>", line 1

```
ax.set_rmax(r + 0.5)
```

```
^
```

IndentationError: unexpected indent

```
>>> ax.set_rmax(r + 0.5)
```

Traceback (most recent call last):

```
File "<stdin>", line 1, in <module>
```

NameError: name 'r' is not defined

```
>>> ax.set_rmax(r + 0.5)
```

Traceback (most recent call last):

```
File "<stdin>", line 1, in <module>
```

NameError: name 'r' is not defined

```
>>> for i, shell in enumerate(elec_structure):
```

```
... #Draw the shell
```

```
... r = 0.5 * (i + 1)
```

```
... rad = [r] * 1000
```

```
... theta = [2 * np.pi * j / 1000 for j in xrange(1000)]
```

```
... ax.plot(theta, rad, 'k-', lw=1)
```

```
... #Draw the electrons
```

```
... rad = [r] * shell[2]
```

```
... theta = [2 * np.pi * j / shell[2] for j in xrange(shell[2])]
```

```
... ax.plot(theta, rad, 'o', markersize=15)
```

```
... rticks.append(r)
```

```
... rlabels.append("{} {}".format(shell[0], shell[1]))
```

```
...
```

```
[<matplotlib.lines.Line2D object at 0x7f2baf37f4d0>]
```

```
[<matplotlib.lines.Line2D object at 0x7f2baf37f7d0>]
```

```

[<matplotlib.lines.Line2D object at 0x7f2baf37fcd0>]
[<matplotlib.lines.Line2D object at 0x7f2baf38d110>]
[<matplotlib.lines.Line2D object at 0x7f2baf38d510>]
[<matplotlib.lines.Line2D object at 0x7f2baf38d950>]
[<matplotlib.lines.Line2D object at 0x7f2baf38dd50>]
[<matplotlib.lines.Line2D object at 0x7f2baf376790>]
[<matplotlib.lines.Line2D object at 0x7f2baf376690>]
[<matplotlib.lines.Line2D object at 0x7f2baf365590>]
[<matplotlib.lines.Line2D object at 0x7f2baf365750>]
[<matplotlib.lines.Line2D object at 0x7f2baf35b4d0>]
[<matplotlib.lines.Line2D object at 0x7f2baf35bd10>]
[<matplotlib.lines.Line2D object at 0x7f2baf35b450>]
>>> ax.set_rmax(r + 0.5)
>>> ax.set_thetagrids([0, 90, 180, 270], [""]*4, color='k')
(<a list of 8 Line2D ticklines objects>, <a list of 4 Text major ticklabel objects>)
>>> ax.set_rgrids(rticks, rlabels)
(<a list of 7 Line2D gridline objects>, <a list of 7 Text major ticklabel objects>)
>>> ax.set_title("Electronic structure of {}".format(symbol), fontsize=20)
<matplotlib.text.Text object at 0x7f2bb0ae0790>
>>> grid(True)
>>> show()
>>>

```