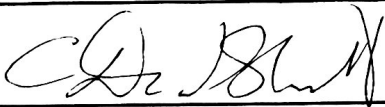



Structured Data and Resource Sharing in Open Source Computational Chemistry

Nicholas Petosa
Dr. David Sherrill, Daniel Smith, Lori Burns
Sherrill Group
Georgia Institute of Technology

Faculty Member 1

Printed: C. David Sherrill
Signed: 

Faculty Member 2

Printed: Edmond Chow
Signed: 

Abstract

When designing drugs and materials, researchers must examine how systems of molecules interact with each other. It is expensive and difficult to execute and measure the results of such experiments in the real world, as there are many physical limitations to providing ideal conditions. The field of computational chemistry attempts to solve this problem by developing advanced software packages such as PSI4 to simulate complicated molecular interactions. These tools take in large amounts of data as parameters and then generate large amounts of data as output. Depending on desired accuracy of results, simulations can take hours to days to run. Sharing this data with other groups is essential for research to move quickly. The volume of data produced by these simulations is difficult for a group to manage, let alone share effectively. The goal of this paper is to discuss the implementation of a computational chemistry data storage and dissemination system so that groups can easily structure data from chemical output for analysis and distribution. By structuring this chemical data rather than just storing it in a flat file, researchers can quickly derive important insights; initial results show that advanced queries for chemical properties of this structured data can execute in around 2 milliseconds. This data storage system introduces a flexible data structuring standard to the field of computational chemistry.

Introduction

Computational chemistry is the intersection of computer science and quantum chemistry. It involves the use of high performance computer simulations to model the behavior, structure, and interactions of molecules. Computational chemistry techniques are used widely to develop new drugs and materials, and are used by chemistry researchers to help confirm theories.

Simulating complex molecular interactions comes at a hefty computational cost, with some simulations taking hours or days to produce accurate results. Researchers must balance computation time and accuracy when running simulations. To mitigate this, computational chemistry software developers focus on optimizing internal algorithms. Despite optimizations in algorithmic execution, the field has neglected optimizing its channels of resource sharing.

There is no agreed upon medium through which computational chemistry datasets are to be distributed to research groups. Goerigk and Grimme distribute their GMTKN (General Main group Thermochemistry, Kinetics, and Non-covalent interactions) databases in a plethora of formats; molecular geometries are distributed as Perl files, and output datasets are available as HTML tables or are appended to their published papers¹. The distribution of unstructured datasets in unstandardized formats has hindered the accessibility of past computational chemistry research. To use published results, groups must manually structure and migrate that data into their own environments, or are forced to run expensive computations all over again.

PSI4 is a popular suite of open-source quantum chemistry programs that can run multi-core parallelized computational chemistry tasks², and coarse-grained parallelization over discrete sub-computations is currently under development. It is maintained by collegiate research groups, with major contributions coming from the Sherrill Group at Georgia Tech. PSI4, like other quantum chemistry programs, can require a large number of parameters (either explicitly provided, or chosen as default values), and the computation can produce large amounts of output data. The purpose of this research is to demonstrate the effectiveness and feasibility of integrating a DBMS (Data Base Management System) into a PSI4 workflow in order to structure, manipulate, and distribute chemical datasets. Ideally, other research groups will

be able to host their PSI4 input and output data on the Sherrill Group's database, or can download the database for in-house research via python's package manager. Ideally, the integration with PSI4 will demonstrate the feasibility and utility of DBMS's in computational chemistry, and will serve as a first step towards a larger software ecosystem that could include other computational chemistry programs. There are no other DBMS integration systems used by the computational chemistry field.

A DBMS solution affords many advantages to individual groups and to the field as a whole, as structured data is much more powerful than flat files or tables. Advanced queries can be used on structured data, such as selecting all molecules having energetic properties within a certain range. A set of methods demonstrating these querying abilities is included. Another benefit is portability; datasets and molecules can easily be exported from the DBMS into a common file format, as well as imported into other environments easily.

Data housed in DBMS environments can easily be manipulated by statistical programming languages such as Python and R, opening the door for advanced data analysis of results¹⁴. Structured data in a DBMS can also easily be connected to a web service for online interaction and dissemination through a centralized server.

Literature Review

The field of computational chemistry concerns the design, implementation, and use of chemical simulations to predict or substantiate theoretical analysis. As the molecular systems required in the design of new drugs and materials continue to grow in complexity, so do the experiments required to study them. Sherrill expresses the view that computational chemistry simulations, particularly those regarding noncovalent interactions (NCIs), hold a significant advantage over traditional experimentation because of a simulation's marked ability to isolate chemical interaction from interference and solvation effects³.

Despite their reliability, simulations of complex molecular bodies are computationally expensive, taking hours to days to execute. One throttle on computational speed is the algorithmic complexity of accurate computations. The most accurate quantum chemistry computations use coupled-cluster theory with single and double substitutions and a perturbative treatment of triple substitutions, CCSD(T), usually referred to as the "gold standard" within the field of computational chemistry⁴, but this approach is computationally expensive. Computational expense can be further exacerbated by the granularity of basis sets. The complete basis set can only be approached, and as the basis set grows, so does computational cost, so a balance must be struck between cost and accuracy⁵.

The substantial time required for each computation makes both algorithm optimization and resource sharing between computational chemistry groups crucial to the advancement of the field. Open source tools such as PSI4 provide groups with the tools for high-performance computational chemistry calculations, providing researchers with access to algorithmically optimized software. Where the field has been suffering is in the ability of groups to access and reuse the results of previously measured chemical systems. The current means of data storage and distribution is remarkably lackluster.

An example of the state of resource management is Goerigk and Grimme's GMTKN30 database. The database is a compendium of chemical subsets that are the results of computational calculations. It is a collection of reference data, with results from in-house calculations as well as from outside groups. For example, Goerigk and Grimme note that the

database was recently updated with Takatani et al.'s revisited calculation of the S22 subset, a subset typically used for benchmarking^{1,6}. GMTKN30 is an attempt to centralize computational chemistry data for resource sharing, and is regularly maintained and updated.

Goerigk and Grimme distribute their GMTKN30 database via website⁷. Molecular geometries are distributed as Perl source code, while reference data and calculation results are distributed as HTML tables on a webpage. For example, the database's page on the S22 subset⁸ provides the following: 57 files, totaling 55.1 kB of serialized Perl objects for molecular geometry, a 22x8 HTML table for reference data, as well as about 50 other HTML tables of variable size to capture results such as GGA density functionals, hybrid density functionals, and double hybrid density functionals. Groups looking to use this shared data must manually structure and migrate that data into their own environment, which takes time and is prone to errors. The chore of porting data becomes either manually entering thousands of lines of data from a flat file or extracting fields from Perl source code by hand, tasks which have been made superfluous with modern database technology. Many computational chemistry repositories which aggregate past results exist, but all structure and distribute their datasets differently. The prevalence of suboptimal distribution slows down the speed of computational chemistry research considerably.

The need for efficient, standardized distribution of structured data is not unique to the field of computational chemistry. Other computational sciences, such as geoscience, similarly require the standardized distribution of data. According to Pons and Masó's research, when structuring map data, geoscience experts struggle to solve the multi-part problem – that is, connecting disparate map parts developed by numerous groups into a continuous, intricate terrain for digital distribution. Their proposed solution was to adopt a field-wide ISO standard to enforce a relational database structure backend and for consistent compression algorithms of API responses to ensure that each node's intricacy would be fully captured while also ensuring compatibility with all other nodes⁹. Computational chemistry lacks the resource sharing infrastructure present in the geosciences. The purpose of this research is to demonstrate a model of structuring and distributing geometries, reference data, and results from computational chemistry calculations.

The backbone of idealized structure and distribution is a DBMS (Data Base Management System) for storing, manipulating, and exporting resulting and reference datasets. The DBMS that will be used for this research is MongoDB¹⁰, and the data that will be structured for experimentation are the S22¹¹ and HBC6¹² subsets. These subsets are used because of their ubiquity in computational chemistry benchmarking, and MongoDB was chosen because it can store records directly from JSON, a format which can intuitively capture the nested structure of PSI4 output.

A DBMS solution affords many advantages, as structured data is much more powerful than flat files or tables. Database operations are fast, as DBMS solutions such as MongoDB are structured with balanced b-trees for $O(\log(n))$ data access¹³. This superior data structuring - enables for advanced queries, such as selecting all molecules with certain properties. Records in the DBMS are exported and distributed in a standard format such as XML (Extensible Markup Language) or JSON (JavaScript Object Notation). As Pons and Masó note, distributing data in a format bound to one DBMS is inappropriate for long term data preservation as the data is not at all decoupled⁹. This issue of high coupling is seen in GTMK30's distribution of Perl source code as a medium for its geometry data. Pons and Masó emphasize that proprietary database table formats should be converted to SQLite, CSV files, or another widespread structured markup language before being incorporated into a package or distributed via web service⁹. Serialization to popular formats such as JSON makes molecular data much more portable.

Data housed in DBMS environments can easily be manipulated in Python or R using database driver libraries. MongoDB has a large breadth of libraries for interfacing with a database instance. This enables statistical computing software such as pandas to access chemical data, opening the door for advanced analysis of statistical error, granularity, and other points of concern. In his conference paper, McKinney discusses other features of pandas that could be applicable to computational chemistry databases, including operations such as pivot, join, and groupby, as well as support for robust regression, panel data, and datetime objects¹⁴.

The goal of this study is to generate a proof-of-concept database exemplifying the ideal structure and distribution of computational chemistry data, as well as developing Python code for querying the database. To do this, results from the S22 and HBC6 subsets will be structured and inserted into a MongoDB DBMS. An interface will then be produced in Python to demonstrate the feasibility of attaching a web server or integrating pandas.

Methods

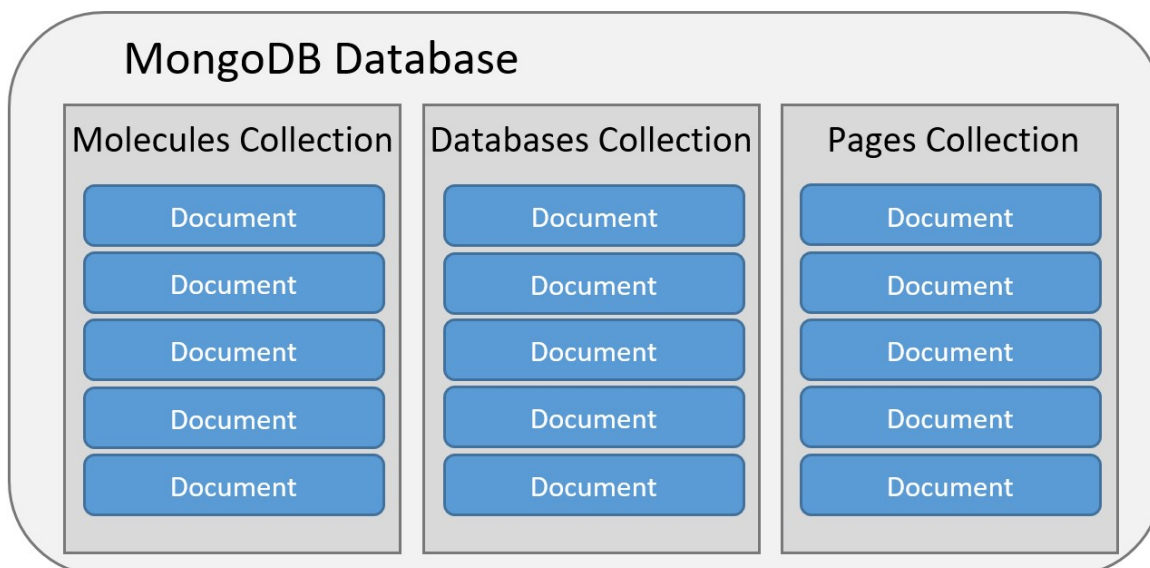
Setting up the environment

The DBMS chosen for implementation was MongoDB v3.2.11 hosted on a Mac Mini running macOS 10.12 (Sierra). MongoDB was selected because chemical outputs contain data that can be well represented by JSON objects, so with a bit of formatting, migration into a Mongo database could be as simple as importing chemical output. The nesting capabilities provided by Mongo documents are convenient for structuring the outputs of chemical simulations, which typically contain data that can be represented by dictionaries inside of dictionaries. In addition, automatic file system snapshots run weekly for redundancy or rollback. A mock database with fake data is also hosted on the Mac Mini for unit testing. This is an open source project; progress and integration tests can be viewed at https://github.com/psi4/mongo_qcdb.

Defining the schema of records

In MongoDB, collections hold data elements called “documents,” which can best be understood as JSON objects. At the highest level of abstraction, this schema defines three collections: molecules, databases, and pages. Each collection contains a specific type of document, and each document has a well-defined schema.

Figure 1 – High-level layout of DBMS.



The molecules collection holds molecules that have been identified for storage in the database. The purpose of this collection is to define a library of molecules that the other collections can refer to by ID. Molecules contain fields such as symbol, masses, name, charge, and geometry. Below is what a JSON record for a molecule object looks like, populated with fake data.

```
{
  "symbols": ["C", "O", "O"],
  "masses": [16.0, 18.0, 18.0],
  "name": "Carbon Dioxide",
  "charge": 0.0,
  "multiplicity": 1,
  "real": [false, false, false],
  "comment": "A test comment",
  "geometry": [
    [3.11, 5.12, 6.14],
    [-3.13, -7.12, -9.18],
    [1.22, 5.11, -1.89]
  ],
  "fragments": [
    [1, 2, 3]
  ],
  "fragment_charges": [0.0],
  "fragment_multiplicities": [1],
  "provenance": {
    "doi": "val",
    "tag": "val",
    "version": "0.7.4alpha0+21.gd658905.dirty",
    "routine": "moldesign.from_smiles",
    "creator": "MolecularDesignToolkit"
  }
}
```

In computational chemistry, there is the concept of a “database,” which does not refer to a DBMS environment but rather to a standard collection of reactions typically used for benchmarking chemical simulations. As previously mentioned, the GMTKN30 is one such database. In the collection, each “databases” document corresponds to a computational chemistry database, such as GMTKN30 or HBC6. Databases contain fields such as name, molecules, and stoichiometry. Below is what a JSON record for a database object looks like, populated with fake data inspired by the HBC6 dataset.

```
{
  "name": "HBC6",
  "reactions": [
    {
      "name": "Fa00Fa00_0.94",
      "stoichiometry": {
        "cp": {
          // Keys are hashes to molecule records.
          "9af98afa": 1.0,
          "c09fb18e": -1.0,
          "f3833784": -1.0
        }
      },
      "reaction_results": ["B97/adz", "BLYP/qzvp", "PBE0/qzvp"],
      "attributes": {
        "R": 0.94
      }
    },
    {
      "name": "Fa00Fa0N_1.31",
      "stoichiometry": {
        "cp": {
          "ef37c9ac": 1.0,
          "09ee42ce": -1.0,
          "32654467": -1.0
        }
      },
      "reaction_results": ["B97/adz", "BLYP/qzvp", "PBE0/qzvp"],
      "attributes": {
        "R": 1.31
      }
    }
  ],
  "provenance": {
    "doi": "val",
    "tag": "val",
  }
}
```

```

    "version": "0.7.4alpha0+21.gd658905.dirty",
    "routine": "moldesign.from_smiles",
    "creator": "MolecularDesignToolkit"
  },
  "citation": "A. Smith and B. Jones",
  "link": "http://example.com"
}

```

The pages collection exists to store data pertinent to a specific molecule and method combination. Without this collection, each database document would have to contain specific information for each molecule that is defined. The document size would increase exponentially and database documents would require constant updates. The pages database solves this problem by decoupling that data. Each page contains values associated to that particular combination of molecule and method, with fields such as molecule, method, value, and type. Below is what a JSON record for a page object looks like, populated with fake data.

```

{
  "molecule_hash": "dbbacd78247e7b39ee5cb8e78d74423e98639203",
  "modelchem": "MP2/aug-cc-pVDZ",
  "return_value": 1.34,
  "type": "energy",
  "success": true,
  "error": "",
  "raw_output": "",
  "options": {
    "opt1": "val",
    "opt2": "val"
  },
  "variables": {
    "var1": "val",
    "var2": "val"
  },
  "provenance": {
    "doi": "val",
    "tag": "val",
    "version": "0.7.4alpha0+21.gd658905.dirty",
    "routine": "moldesign.from_smiles",
    "creator": "MolecularDesignToolkit"
  }
}

```

Implementing a wrapper

The aforementioned schema must be enforced by a software layer that interacts with the database. In order to work with the database, JSON files encapsulating chemical records are used as intermediaries between PSI4 and the DBMS environment. The software wrapper can

interface between JSON files on disk and the MongoDB instance in order to add documents or perform advanced queries.

Part of the wrapper's job is to compute SHA1 hashes for all inbound JSON objects. This hash is used as the `_id` of the document instead of MongoDB's default ObjectID. This unique ID attached to each document added to the database prevents duplicate entries and increases access speed dramatically. SHA1 hashes are superior to ObjectIDs because the SHA1 hash is reflective of the actual content of the document. Hence, it is persistent through database flushes and can be calculated independently of DBMS environment, whereas an ObjectID would be reset if a document is removed and re-added. Computing a SHA1 hash of the fields that define uniqueness is a much cleaner solution and introduces a standardized hashing practice to computational chemistry for naming specific molecules, pages, and databases. Hashes are used instead of indexing on multiple fields because MongoDB does not support indexing of dictionary fields like geometry.

For all document types, we only hash a few essential fields as opposed to the entire document. This allows for small changes to the JSON during production without the need to recalculate the entire hash. Those specific fields that are hashed are explicitly defined.

Hashed fields for molecule records:

```
"symbols": ["C", "O", "O"],
"masses": [16.0, 18.0, 18.0],
"name": "Carbon Dioxide",
"charge": 0.0,
"multiplicity": 1,
"real": [false, false, false],
"geometry": [ . . ],
"fragments": [ . . ],
"fragment_charges": [0],
"fragment_multiplicities": [1]
```

Hashed fields for database records:

```
"name": "S22"
```

Hashed fields for pages:

```
"molecule": "dbbacd78247e7b39ee5cb8e78d74423e98639203",
"method": "MP2/aug-cc-pVDZ"
```

This package is designed to expose several important querying functions. Listed below are the functions implemented and a brief description of what they do.

def add_generic(self, data, collection): Given **data**, which is a molecule, database, or page in JSON format, adds it to the respective collection specified by **collection**. If a record with the same hash exists already, a duplicate key exception is thrown and the document is not added.

```
def add_generic(self, data, collection):
    sha1 = fields.get_hash(data, collection)
    try:
        data["_id"] = sha1
        self.project[collection].insert_one(data)
        return True
    except pymongo.errors.DuplicateKeyError:
        return False
```

def evaluate(self, hashes, methods, field="return_value"): Queries mongod for all pages containing a molecule specified in **hashes** and all methods specified in **methods**. For all matches, finds their **field** value and populates the relevant dataframe cell.

```
def evaluate(self, hashes, methods, field="return_value"):
    hashes = list(hashes)
    methods = list(methods)
    command = [{"$match": {"molecule_hash": {"$in": hashes}, "modelchem": {"$in":
methods}}}]
    pages = list(self.project["pages"].aggregate(command))
    d = {}
    for mol in hashes:
        for method in methods:
            d[mol] = {}
            d[mol][method] = np.nan
    for item in pages:
        scope = item
        try:
            for name in field.split("."):
                scope = scope[name]
            d[item["molecule_hash"]][item["modelchem"]] = scope
        except KeyError:
            pass
```

```
return pd.DataFrame(data=d, index=[methods]).transpose()
```

def evaluate2(self, hashes, fields, method): Queries mongod for all pages containing a molecule specified in **hashes** of method **method**. For all matches, finds the values in each of their **fields** and populates the relevant dataframe cell.

```
def evaluate_2(self, hashes, fields, method):
    hashes = list(hashes)
    command = [{"$match": {"molecule_hash": {"$in": hashes}, "modelchem": method}}]
    pages = list(self.project["pages"].aggregate(command))
    d = {}
    for mol in hashes:
        for field in fields:
            d[mol] = {}
            d[mol][field] = np.nan
    for item in pages:
        for field in fields:
            scope = item
            try:
                for name in field.split("."):
                    scope = scope[name]
                d[item["molecule_hash"]][field] = scope
            except KeyError:
                pass
    return pd.DataFrame(data=d, index=[fields]).transpose()
```

def search_qc_variable(self, hashes, field): Returns a pandas dataframe with your results. The rows will have the molecule **hashes** and the column will contain the name. Each cell contains the **field** value for the molecule in that row.

```
def search_qc_variable(self, hashes, field):
    d = {}
    for mol in hashes:
        command = [{
            "$match": {
                "molecule_hash": mol
            }
        }, {
            "$group": {
                "_id": {},
                "value": {
```

```

        "$push": "$" + field
    }
}
}]
pages = list(self.project["pages"].aggregate(command))
if (len(pages) == 0 or len(pages[0]["value"]) == 0):
    d[mol] = None
else:
    d[mol] = pages[0]["value"][0]
return pd.DataFrame(data=d, index=[field]).transpose()

```

def list_methods(self, hashes): Returns a pandas dataframe with your results. The rows will have the molecule **hashes** and the columns will be numbered. Each cell contains a method used by the molecule in that row.

```

def list_methods(self, hashes):
    d = {}
    for mol in hashes:
        records = list(self.project["pages"].find({"molecule_hash": mol}))
        d[mol] = []
        for rec in records:
            d[mol].append(rec["modelchem"])
    df = pd.DataFrame(dict([(k, pd.Series(v)) for k, v in d.items()])).transpose()
    return df

```

Also included in the python wrapper are two functions for pushing database records over a network. The code for those operations as well as the wrapper in its entirety are in the full codebase.

Results

All measurements ran from a locally hosted MongoDB instance.

Figure 2 – Real-world runtime analysis

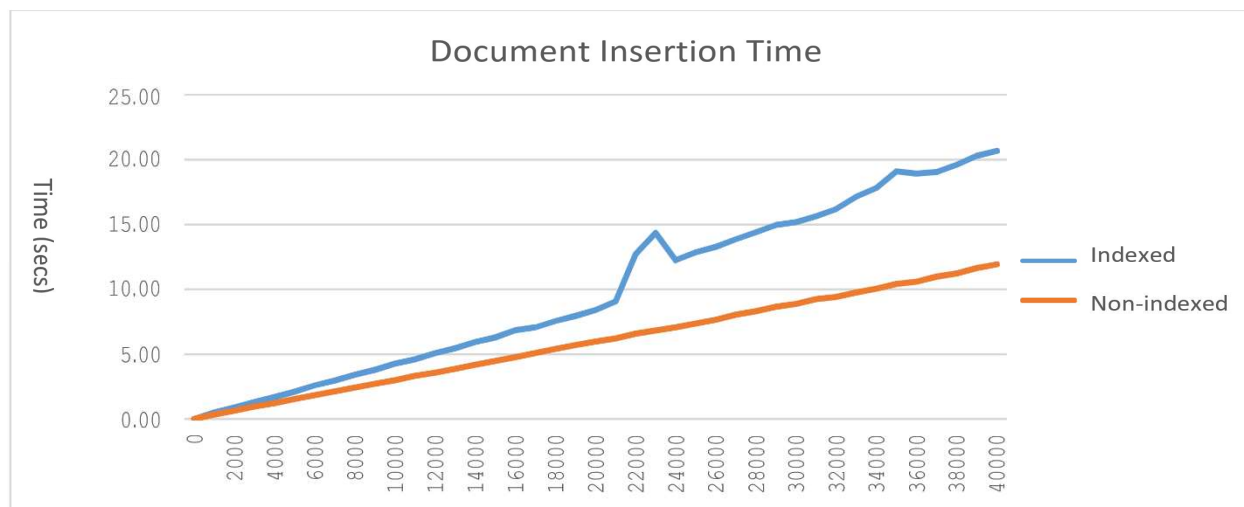
Contents of database – total of 930 items.

Subset / Collection	Molecules Collection	Databases Collection	Pages Collection
S22	110 items	1 item	110 items
HBC6	354 items	1 item	354 items

Function runtimes

Aggregation Function	Time (ms) (Average of 1000 runs)	Parameters
constructor	560	-
evaluate	1.5	2 molecules, 2 methods
evaluate_2	1.8	2 molecules, 2 fields
search_qc_variable	2.1	2 molecules
list_methods	1.5	2 molecules

Figure 3 – Populated database with 40,000 fake records and compare insertion time when indexing is enabled versus disabled.



Discussion

The results of this test run suggest that we have designed and implemented an efficient infrastructure for housing quantum chemistry datasets. As seen in figure 1, the aggregation functions take 1ms to 2ms and are incredibly fast. Similarly, figure 2 shows that data insertions take constant time since time taken increases linearly with number of documents added. The blue line represents the actual insertion time with our current implementation, where each record is labeled with a custom index that we calculate and provide to MongoDB. The orange line was calculated by removing all hash calculation from our python code and allowing MongoDB to assign each record with a random, internally generated Object ID. Each indexed insertion takes an average of 0.45ms, while each non-indexed insertion takes an average of 0.3ms, so insertions are 50% slower when indexed. There is a small spike in our indexed insertion time at 21000 documents and 35000 documents, but this can be attributed to random hardware fluctuations, as the overall slope is overwhelmingly constant and positive.

Runtime Analysis

From the exceptional performance results of our aggregation functions, we confirm that indexing our records had increased access time dramatically. Typically, accessing an element in a B-tree like those used in MongoDB would take $O(\log n)$ time, but indexing our elements has enabled constant-time lookup of our data. However, this optimization does cost us in insertion time. From our results, we see that most bottlenecking will be the result of insertion latency during database construction (560ms). Since MongoDB must check for document collisions as well as add our hashes to an internal map, inserting documents with a custom index adds substantial overhead to the insertion cost of each piece of data. Additionally, we must calculate the hash of each piece of data added the DBMS, which takes a considerable amount of time.

For our use case, it is far more important to minimize access time than insertion time. Documents are only inserted once, whereas accesses happen every time a user calls one of our aggregation functions. This is especially important once the database is exposed to a network connection and an entire community has access to these lookup functions. However, it is still important to consider ways of reducing the time cost of insertions if we can keep the time cost of accessing data the same. When backing up the DBMS, the `clone_to` and `copy_to` functions need to rebuild an entire database, so insertion cost is relevant beyond inception.

One possible way of reducing insertion time is to use a simpler hash function than SHA1 to calculate checksums. A simpler hashing function runs the risk of hash collisions but will also run fast.

Limitations

This test does have a few limitations. There will be magnitudes more data in the DBMS once other datasets are migrated into the environment. Although our schema should not have any issues, the implications of increased volume on insertion and access runtime is unknowable until we can fully populate our database. For example, there may be access and insertion slowdown as the index grows. Another unknown is how increasing the number of parameters will affect speeds. Ideally, more benchmarking datasets will be added to this database so we can measure how performance scales with more parameters. This would confirm that we have truly achieved constant-time access in our aggregation functions. It may be the case that users experience unexpected slowdown as they increase the number of molecules or methods they are querying for, as record parsing and formatting is handled on the slower python end. That said, the provided benchmarks for 2 argument calls should be the standard use case for this domain and provide sufficient insight into operating speeds for common calls.

Future Work

More research should be done to find a hashing function that is cheap to run yet robust enough for this domain. Another possibility is to restructure our records so that there are enough simple key-value fields in each to use MongoDB's built-in multi-field indexing capabilities. Without a doubt, MongoDB's built in indexing logic will run faster than our Python hashing library. Earlier we decided to us SHA1 hashing because multi-field indexes cannot index dictionary fields, so if we can somehow represent our data without nested dictionary structures, we could take advantage of multi-field indexing.

Now that a structured DBMS is in place, we can expose our data via web server for network access. Currently a Tornado web application is being developed on this project's open source repository to wrap these low-level functionalities. This web server will also include Dask integration for distributed computation. In-house tests currently have this stack fully functional on a local network, including a distributed scheduler which distribute work between multiple nodes. Once complete, this code will be available for download through python's package manager as `mongo_qcdb`.

Conclusions

A unified storage and distribution system for the computational chemistry community would help accelerate discovery in chemistry and adjacent fields. As mentioned, the computational chemistry community has neglected optimizing resource sharing and data storage. There are no other DBMS software packages specialized for this field. Offering a pre-packaged solution lowers the technical barrier for entry, while at the same time the flexible design of components such as selective field hashing affords extensibility and modularity. The software package will easily be downloadable via python's package manager and then easily integrated into the PSI4 workflow. The schema-less Mongo backend allows for dramatic structural changes if required by the community, and output can easily be exchanged via web API or other means, as JSON is a widely used standard. The goal of the project is for the community to be introduced to a possible standard for chemical data storage, and for that standard to change organically to fit the nuances of computational chemistry, including the development of interfaces to other popular quantum chemistry programs besides PSI4.

References

- [1] L. Goerigk and S. Grimme, "Efficient and Accurate Double-Hybrid-Meta-GGA Density Functionals Evaluation with the Extended GMTKN30 Database for General Main Group Thermochemistry, Kinetics, and Noncovalent Interactions," *Journal of chemical theory and computation*, vol. 7, pp. 291-309, 2010
- [2] R. M. Parrish, L. A. Burns, D. G. A. Smith, A. C. Simmonett, A. E. DePrince, E. G. Hohenstein, U. Bozkaya, A. Y. Sokolov, R. D. Remigio, R. M. Richard, J. F. Gonthier, A. M. James, H. R. Mcalexander, A. Kumar, M. Saitow, X. Wang, B. P. Pritchard, P. Verma, H. F. Schaefer, K. Patkowski, R. A. King, E. F. Valeev, F. A. Evangelista, J. M. Turney, T. D. Crawford, and C. D. Sherrill, "Psi4 1.1: An Open-Source Electronic Structure Program Emphasizing Automation, Advanced Libraries, and Interoperability," *Journal of Chemical Theory and Computation*, vol. 13, no. 7, pp. 3185–3197, Jun. 2017.
- [3] C. D. Sherrill, "Computations of Noncovalent Pi Interactions," *Rev. Comput. Chem*, vol. 26, 2008.
- [4] K. Raghavachari, G. W. Trucks, J. A. Pople, and M. Head-Gordon, "A fifth-order perturbation comparison of electron correlation theories," *Chemical Physics Letters*, vol. 157, no. 6, pp. 479–483, 1989.
- [5] J. RAezáč and P. Hobza, "Describing noncovalent interactions beyond the common approximations: How accurate is the "gold standard," CCSD (T) at the complete basis set limit?," *Journal of Chemical Theory and Computation*, vol. 9, pp. 2151-2155, 2013.
- [6] T. Takatani, E. G. Hohenstein, M. Malagoli, M. S. Marshall, and C. D. Sherrill, "Basis set consistent revision of the S22 test set of noncovalent interaction energies," *The Journal of chemical physics*, vol. 132, p. 144104, 2010.
- [7] "The GMTKN30 database", Thch.uni-bonn.de, 2016. [Online]. Available: <http://www.thch.unibonn.de/tc/downloads/GMTKN/GMTKN30/GMTKN30main.html>. [Accessed: 05- Nov- 2016].
- [8] "The GMTKN30 database - The S22 subset", Thch.uni-bonn.de, 2016. [Online]. Available: <http://www.thch.uni-bonn.de/tc/downloads/GMTKN/GMTKN30/S22.html>. [Accessed: 05- Nov- 2016].
- [9] X. Pons and J. Masó, "A comprehensive open package format for preservation and distribution of geospatial data and metadata," *Computers & Geosciences*, vol. 97, pp. 89-97, 2016.

- [10] "MongoDB for GIANT Ideas," MongoDB. [Online]. Available: <http://www.mongodb.com/>. [Accessed: 04-Feb-2018].
- [11] P. Jurečka, J. Šponer, J. Černý, and P. Hobza, "Benchmark database of accurate (MP2 and CCSD(T) complete basis set limit) interaction energies of small model complexes, DNA base pairs, and amino acid pairs," *Phys. Chem. Chem. Phys.*, vol. 8, no. 17, pp. 1985–1993, 2006.
- [12] K. S. Thanthiriwatte, E. G. Hohenstein, L. A. Burns, and C. D. Sherrill, "Assessment of the Performance of DFT and DFT-D Methods for Describing Distance Dependence of Hydrogen-Bonded Interactions," *Journal of Chemical Theory and Computation*, vol. 7, no. 1, pp. 88–96, Nov. 2011.
- [13] C. Jandaeng, "Comparison of RDBMS and document oriented database in audit log analysis," in 2015 7th International Conference on Information Technology and Electrical Engineering (ICITEE), 2015, pp. 332-336.
- [14] W. McKinney, "Data structures for statistical computing in python," in Proceedings of the 9th Python in Science Conference, 2010, pp. 51-56.