Community Accessible Datastore of High-Throughput Calculations: Experiences from the Materials Project

Dan Gunter, Shreyas Cholia, Anubhav Jain,
Michael Kocher, Kristin Persson, Lavanya Ramakrishnan
Lawrence Berkeley National Laboratory
Berkeley, CA 94720, USA
{dkgunter,scholia,ajain,mpkocher, kapersson,lramakrishnan}@lbl.gov

Shyue Ping Ong, Gerbrand Ceder
Massachussetts Institute of Technology
Cambridge, MA 02139, USA
{shyue,ceder}@mit.edu

***Abstract*—Efforts such as the Human Genome Project provided a dramatic example of opening scientific datasets to the community. Making high quality scientific data accessible through an online database allows scientists around the world to multiply the value of that data through scientific innovations. Similarly, the goal of the Materials Project is to calculate physical properties of all known inorganic materials and make this data freely available, with the goal of accelerating to invention of better materials. However, the complexity of scientific data, and the complexity of the simulations needed to generate and analyze it, pose challenges to current software ecosystem. In this paper, we describe the approach we used in the Materials Project to overcome these challenges and create and disseminate a high quality database of materials properties computed by solving the basic laws of physics. Our infrastructure requires a novel combination of high-throughput approaches with broadly applicable and scalable approaches to data storage and dissemination.**

I. Introduction

Materials discovery and development is a key innovation driver for new technologies and markets, and an essential part of the drive to a renewable energy future. Yet, historically, novel materials exploration has been slow and expensive, taking on average 18 years from concept to commercialization. [8] To address this challenge, the US government has created the US Materials Genome Initiative (MGI) [18], which aims to "double the speed with which we discover, develop, and manufacture new materials".

The central component of the MGI approach is using our ability to accurately model nature through computer simulations at unprecedented scale. It is now well established, through several demonstrated examples [11], that many materials properties can be predicted by computing accurate approximate solutions to the basic laws of physics, and that this virtual testing of materials can be used to design and optimize materials *in silico*. By applying the power of *many-task computing* [26] on increasingly powerful computational platforms, materials designers, both theorists and experimentalists, can scan through thousands of possible new materials across a wide range of chemistries.

The Materials Project (MP) [19], part of and in fact a progenitor of the MGI, is providing a community accessible datastore of high-throughput calculations that scientists can leverage to quickly predict, screen, and optimize materials for target properties. A major goal of MP is to populate this community datastore with calculated properties of all known inorganic materials.

The Materials Project (MP) has benefited from HPC resources at NERSC and elsewhere, consuming roughly 8 million CPU hours to date. Many-task computing workflows are increasingly using HPC environments due to their need for large computation and storage resources. HPC environments present challenges for running both the datastores and the associated calculation workflows because these environments were originally designed to serve the needs of large MPI applications that run for predictable times and do all I/O to disk. Many-task workflows such as the Materials Project have low parallelism with sometimes very unpredictable total runtimes, and these workflows need to update a database with their results.

Contributions. The two major contribution of this paper are, first, to describe how our infrastructure uses a NoSQL datastore for materials properties as a central component that serves multiple roles: (a) managing the state of high-throughput calculations (performed by our high-throughput workflow engine), (b) storage and analytics for the calculation results, and (c) a searchable back-end for data dissemination. We have leveraged the flexibility and scalability of a NoSQL "document store" [3], MongoDB, to achieve these three goals within the same deployment. The second contribution is to describe, from our experiences, the current challenges in deploying any centralized datastore of this type within the HPC ecosystem.

II. Related Work

CatApp, developed by Hummelshoj et al. [13] is an example of high-throughput materials design that provides a web application to access activation energies of elementary surface reactions and is part of a larger database of surface reaction data being developed under the Quantum Materials Informatics Project [25]. Curtarolo et al. [5] have developed

978-0-7695-4956-9/13 $26.00 © 2013 IEEE
DOI 10.1109/SC.Companion.2012.150

the AFLOW (Automatic Flow) software framework for high-throughput calculation of crystal structure properties of alloys, intermetallics and inorganic compounds. The Materials Project is much more user-centric and application-agnostic than these efforts, i.e. the primary focus is not a researcher screening for an application, but rather enabling the user to access to the data in a way that lets them discover applications.

Other open databases built from surveys of nature include the Human Genome Project (HGP), whose goals were to identify, store, and disseminate all the approximately 20,000-25,000 genes and 3 billion chemical base pairs in human DNA. The Sloan Digital Sky Survey (SDSS) [28], over eight years of operations, obtained deep, multi-color images covering more than a quarter of the sky, mapping more than 930,000 galaxies and more than 120,000 quasars. The Materials Project goals are similar in breadth, but focus more heavily on computation as a generator of data and on a community-owned library of analysis tools.

Other examples of collaborative data-centric portals include the Earth System Grid (ESG) [31], [9], which focuses on climate and environmental science data sets, in particular for the World Climate Research Programme's Coupled Model Intercomparison Project. The Systems Biology Knowledgebase (KBase) [16] is a collaborative effort designed to accelerate our understanding of microbes, microbial communities, and plants, by providing free and open access to data, models and simulations. Both these projects differ from MP due to very different requirements: the ESG has to enable exchange of huge data sets, as opposed to MP's finer-grained collaborative data sharing; KBase must enable a cross-domain exploratory process but does not concern itself like MP with ab-initio calculations as a common exploratory base.

III. Design and Implementation

The current MP implementation is based on the high-throughput framework developed by Jain et al. [14] and subsequently extended by collaborators at the Lawrence Berkeley Laboratory and National Energy Research Scientific Computing Center (NERSC). The framework has performed computations to screen over 80,000 inorganic compounds for a variety of applications, including Li-ion and Na-ion batteries. [4], [10], [12], [20], [22].

This computational infrastructure was created to discover new, better, materials using high throughput *ab initio* computations. Currently the focus is on atomic scale calculations of thermodynamic and electronic properties using density functional theory, but the methodology is applicable to many different length scales, properties, and methods.

An example of the investigations enabled by this infrastructure is the search for better lithium-ion battery materials, which would make many common devices perform better and impact the environment less. Two crucial properties for any battery are its voltage and capacity. In Figure 1, we show potential battery materials screened by the Materials Project as a function of predicted voltage and capacity, noting the comparatively narrow range of properties exhibited by known materials. Several compounds displayed in Figure 1 could improve upon the properties of known materials; further computations can be used to screen promising candidates for other important properties such as Li diffusivity (related to power delivered by the cell) Without high-throughput computing, theorists and experimentalists need to choose which compounds to compute and synthesize based on a combination of extrapolation and intuition that is far slower and may missing promising candidates.

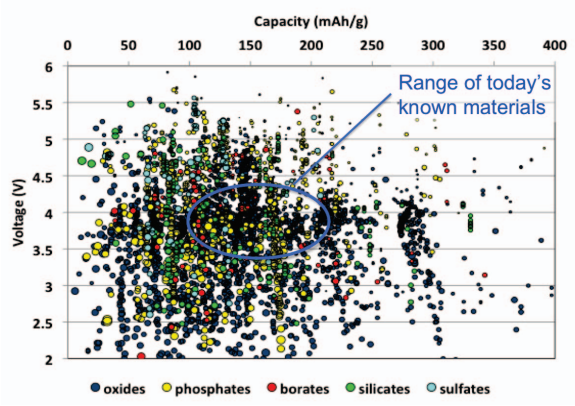

Fig. 1: Battery materials screened.

The current Materials Project implementation runs as a 24/7 production system with, as of this writing, over 2500 registered users, many of whom are very active: for example, in the week of August 20 - 27, 2012 the web interface logged 3315 distinct queries returning a total of $12,951,099$ records.

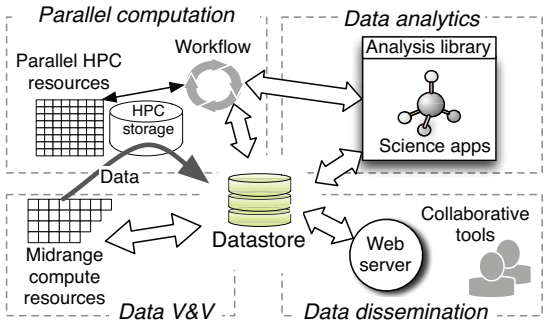

Fig. 2: Materials Project architecture. The datastore serves all four major functions, clockwise from upper-left: Parallel computation, Data analytics, Data dissemination, and Data validation and verification.

A. Architecture

The MP architecture, shown in Figure 2, is centered on a flexible, scalable datastore. The computations are driven by the workflow engine, which persists its state in the datastore. The results of the computations are stored on the HPC resource, then loaded into the datastore by midrange compute resources, which also run validation and verification functions. The data

is disseminated through a web server that implements both a Web UI and Web API; collaborative tools allow users to publicly annotate the data. Data analytics and scientific applications are shared through an open-source Python library. Most of these capabilities are available in some form in other HTC systems, but our approach is unique in that all these components coordinate through the datastore, which simultaneously acts a message queue, analytics engine, and web back-end DB.

The envisioned role of the Materials Project (MP) infrastructure in the scientific discovery process is shown in Figure 3. The scientific user begins with ideas (a), which may stem in part from data mining of the MP database. From these ideas, the user creates an initial set of materials to submit for computation. The materials are serialized into records in the Materials Project Source (MPS) format (b).

The MPS records will be converted to a job for the MP workflow (c), which will compute the desired physical properties on parallel resources. Up to this point, all inputs have been generated by the core MP team, i.e. from the ICSD database, or from personal interaction with other scientists. The vision, however, is to open this process to a much broader base of users.

The resulting data can be uploaded to a user-controlled area called a *sandbox* (d), which is only visible to the creator and selected collaborators. This capability is not yet developed, so currently ther is only one "core" database. This core, vetted, database is highly valuable and will continue alongside the private user sandboxes.

The user will analyze the data (e), using the open analytics platform *pymatgen*, to determine the stability and for synthesis potential of the new materials. This will either result in novel materials, or generate new ideas and restart the process. At any point (*e.g.*, after a publication or a patent filing), the user can allow the data to become publicly disseminated through the MP website (f) to the broader community. We note that this capability will be a natural by-product of the Web UI for the sandboxes.

The rest of this section will describe the design and implementation of the enabling infrastructure components: the datastore, the workflow, and the web APIs.

B. Datastore

This section describes how we used document-oriented NoSQL datastore to act as back-end repository, execution engine, and workflow manager. The datastore handles a wide variety of data types, including execution state, outputs, and views of the calculated material properties. We chose NoSQL primarily for flexibility: unlike RDBMS products such as MySQL and PostgreSQL, our datastore does not require that we lay down a normalized schema between all these data types at the beginning of the project. The data being stored is continually evolving as we add new types of calculations and collaborators onto the project over time. By choosing NoSQL, MP can adapt quickly to these changes with small changes in Python code instead of refactoring complex relational schemata.

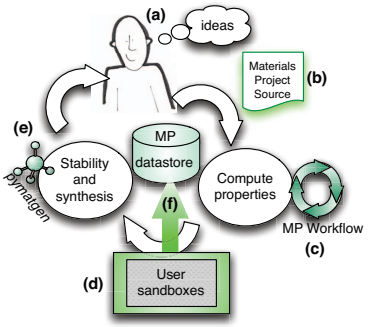

Fig. 3: Envisioned materials discovery workflow. User ideas (a) for candidate materials (b) are submitted for computation (c), stored in user sandboxes (d), analyzed (e), and eventually released to the public (f).

TABLE I: Complexity and structure of selected collections

Collection	Summary	Structure
Battery prototypes	Nodes: 14 Depth: 4 Mean depth: 3.6	
Materials Project Source (MPS)	Nodes: 94 Depth: 6 Mean depth: 4.8	
Materials	Nodes: 208 Depth: 10 Mean depth: 6.0	
Tasks	Nodes: 1077 Depth: 12 Mean depth: 7.4	

While the VASP calculations are running, they generate from a small input (the initial crystal) several MB of intermediate output data. This is parsed and reduced by the FireWorks Analyzer discussed in §III-C3, so that the aggregate volume of data stored in our database remains relatively small, in the hundreds of GB. This data is, though, highly complex: we store hundreds of fields describing calculations for over $30,000$ materials, $3,000$ bandstructures, 400 intercalation batteries, and $14,000$ conversion batteries. An overview of the complexity of the document structures is illustrated as graphs in Table I. Both the web interface and workflow components perform complex ad-hoc queries over these structures.

Our datastore of choice is MongoDB. Among document-oriented datastores, MongoDB is known for its powerful but simple query language, ease of administration, and good performance on read-heavy workloads where most of the data can fit into memory. Its relative weakness for huge datasets

and write-heavy workloads is a reasonable trade-off for MP. A productivity benefit of MongoDB is that both the query language and the native data model is JSON, which is the standard data format for modern web applications and easily represented and manipulated as native Python `dicts`.

The remainder of this section describes the organization of the datastore to serve multiple overlapping roles. In MongoDB terminology, we describe the different *collections* to hold records of similar type, called *documents*.

1) Input data: The input data is our standard JSON representation of a crystal and its metadata, called Materials Project Source (MPS); it may come from a user or an external data source. Our initial dataset was populated from the crystal structures in the Inorganic Crystal Structure Data (ICSD) database [1], a standard dataset in the field. Essential information that must be stored and accessed is standard physical characteristics (atomic masses, positions, etc.), and metadata indicating the source of the crystal.

The input data are stored in the `mps` collection. Because MongoDB and MPS are both JSON, import and export of the data is trivial.

2) 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.

We store all the execution state 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 that resembles Condor *classads* [27], but which can operate on the attributes of the input data directly. For example, to select jobs for crystals containing both lithium and oxygen atoms with less than 200 electrons, we send the query: `{elements:{$all:['Li','O'], nelectrons:{$lte:200}}}`

The `tasks` collection is used for completed tasks and all the associated results. This collection contains much more robust data about the output state and data produced by the calculation. Different task documents can represent different versions of VASP and other codes side by side.

3) Calculated property data.: There are several types of calculated properties that must be stored, including 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. It turns out that the definition of what makes a "material" is tricky: there may be multiple results in `tasks` corresponding to the same MPS input. We wish to present only one result to the user, so we run a MapReduce operation on the `tasks` to group them by the MPS identifier and pick a single "best" result. This process of selection, grouping, and projection is performed in Python code.

4) Summary: In this section we described our use of a document store database, MongoDB, to flexibly handle varied data.

One concern with schemaless databases is that changes in the data layout may go undetected until run-time, where they can introduce complex bugs. To address this, we have created Python classes that abstract database operations, and use these as an intermediate layer. The intermediate layer also provides a defense against lock-in to MongoDB's query language, as it is capable of transforming input queries and updates for another datastore.

C. Workflow

Scientific workflow tools are used to specify dependencies and execute tightly coupled to many-task workflows on HPC systems [7], [30]. Early in the project, we evaluated a number of these tools to assess their applicability to our needs. We identified a number of gaps and challenges:

- *Programmability*. The MP runs a C++ framework (AFLOW) with Python scripts, and we wanted to leverage this knowledge instead of learning details of tool-specific graphical interfaces and/or DSL's.
- *Administration overhead*. A database and web server were already required to serve users. Running another complex persistent service, as required by a majority of these workflow tools, was unwanted overhead.
- *Flexibility*. Existing workflow tools have little support for interacting with databases or reconfiguring running workflows based on application results.

1) Workflow Description: We currently perform computations with the Vienna Ab-initio Simulation Package (VASP) [17], [29]. VASP computes an approximate solution to the many-body Schrödinger equation using Density Functional Theory (DFT). The primary result of this computation is a determination of the charge density for a given compound, or *crystal*. Many useful properties of materials formed from these crystals can be derived directly from this result.

The DFT calculation performed by VASP is complex and challenging to schedule. The core method is really a series of algorithms, each of which is an iterative calculation with several key parameters. There is no single set of parameters or iterative algorithms that works best for all types of crystals, and there is no guarantee that a given run will converge at all. The runtime of a given calculation can be estimated based on domain knowledge, but there is a high degree of uncertainty to

this estimation. The absolute range of runtimes, at the current parallelism allowed by VASP, ranges from minutes to days.

The VASP DFT calculations impose unique requirements on the system that motivate our workflow design.

2) FireWorks: FireWorks is our custom workflow manager and is implemented in Python. Python is a commonly used language in this community and thus reduces the learning curve for future users who might want to write and/or change existing workflows.

A *Firework* represents one step in a workflow, and can consist of several sub-components that are used to define both typical job execution and exception handling. Each job, that runs on the HPC system, is specified as a dictionary of runtime parameters (*Stage*) that are later translated into input files on a compute node by a component called the *Assembler*. The job specification blueprint and subsequent translation to execution state (i.e., input files) by the *Assembler*, is dependent on the desired code to be executed. Because the job specifications are Python dicts, they are easily stored and queried as JSON documents in MongoDB.

In addition to job specification, each Firework also contains components that allow for planned dynamic elements within the workflow. A *Fuse* object is embedded within each Firework and is capable of overriding input parameters prior to execution, based on the output state of any parent jobs. The parameters to override are specified as a Python dict that is similar to Mongo atomic update syntax (e.g. $set, $unset, etc.). This allows any modifications returned by the Fuse to be stored within the FireWorks database for later analysis.

FireWorks has unique features to handle restarting failed jobs, changing parameter sets based on job behavior, and preventing duplicate job submission from multiple users, that we discuss in the next section.

3) FireWorks Unique Features: The Materials Project workflows helped us identify four unique requirements for FireWorks:

- **Re-runs**. All workflow systems must deal with machine failures. For MP, the difficulty in predicting resource requirements means that jobs are also often killed due to insufficient walltime and memory. This needs to be efficiently detected and jobs restarted, *with more resources*.
- **Detours**. MP jobs will sometimes quit with an error message. In these cases, the job must be resubmitted, but with a few minor input parameters changed. The rest of the workflow should be the same. There might be several iterations of modifications before the job runs to completion. If the problem is beyond automated repair, the system needs to abort the entire workflow and mark it for manual intervention.
- **Duplicate detection**. Calculations performed as part of a workflow may have (practically) identical jobs. Since duplicates exist for a large percentage of potential jobs, the system needs to avoid re-running these jobs. Duplicates may arise from two users simply submitting the same thing, or from a job that was specified dynamically during the running of a workflow.
- **Iteration**. Some calculations require iterative runs of the same job, with incrementing input parameters, until a condition is met. In general, the number of iterations required is not known in advance. More sophisticated search algorithms than simple linear increments (*e.g.*, genetic algorithms) may be required.

We believe that these requirements are applicable to many other iterative multi-step calculations, such as quantum Monte Carlo and molecular dynamics.

The *Fuse* object handles delayed execution through conditions (e.g., parent jobs have finished, the parent jobs have some specific output value, a user has approved the workflow, etc.) Because each Fuse is essentially an instance of a Python class, the logic can be arbitrarily complex and specified directly in the programming language.

In addition to the Fuse, dynamic workflows are also managed by embedding an Analyzer object into the FireWork. The Analyzer contains Python code that is run after job completion, and can check jobs then schedule follow-up actions. For example, to perform **re-runs** with jobs that have failed due to insufficient walltime, the Analyzer can create a new Firework that is a copy of the failed job but with a longer walltime. To handle **detours**, the Analyzer can terminate a workflow, or create an entirely new workflow based on the result of the job.

Duplicate jobs are detected via *Binder* objects, which uniquely identify a job. In the case of VASP runs, a Binder may contain a reference to a crystal structure ID and the type of functional (e.g., GGA). FireWorks uses the uniqueness defined by the Binder to replace the execution of duplicate jobs with a pointer to the previous result. By defining appropriate Binders, the FireWorks code allows workflows to be idempotent and be submitted without regard to prior history of the project.

Although **iteration** of the VASP calculations could be constructed from the Fuse, Analyzer, and Binder components, the current implementation uses the previously mentioned AFLOW framework for this inner loop.

In conclusion, FireWorks provides a highly dynamic system for specifying, in Python code, complex rules for choosing and running MP jobs. The time to load the full results of codes is significant and discussed in §IV-C1. Aside from that, system overheads are minimal. The queries to pull down inputs and update the database with new job statuses execute in a negligible fraction of the time to perform the calculations.

D. Data dissemination

The materials discovery process shown in Figure 3 depends on a comprehensive and extensible system for data dissemination. This section describes the Web user interface, programmatic data access, and collaborative tools used in the Materials Project. Our data dissemination component exemplifies the interfaces appropriate for wider data dissemination from community databases.

1) Web user interface: Well-designed web interfaces are important for creating productive and usable environments that enable data-driven science.

The Materials Project places a strong emphasis on user experience and user interface design. We have built a rich, interactive web portal focusing on the scientist as the end-user. Our interface uses technologies like HTML5 and AJAX to allow users to search and browse MP data and pan and zoom real-time visualizations of bandstructures, diffraction patterns, and other properties.

2) Programmatic data access: The web interface makes it very easy for users to interact with the data but there is also a need for programmatic interfaces to build higher-level tools. To provide an open programmatic platform for accessing Materials Project data, we provide a HTTP-based API called the *Materials API*, which is a Web API that maps HTTP URIs to data objects and functions. Results are returned in JavaScript Object Notation (JSON) format [15] that can easily be consumed by other software for processing and analysis. The following example URI shows the simplicity of the Materials API. The example in Figure 4 shows how to retrieve the calculated energy of ferric oxide (Fe_2O_3).

Fig. 4: Materials API URI to get the energy of Fe_2O_3

3) Open analytics platform: pymatgen: One of the goals of MP is to enable users to build a rich set of tools to analyze materials data. Towards this end we provide an open-source Python library called *pymatgen* [23], [24], which defines a Python object model for materials data along with a well-tested set of structure and thermodynamic analysis tools to act on the data. The *pymatgen* library can import and export data from a number of existing formats, including fetching data via the Materials API. This provides a natural and powerful interface for jointly analyzing local and remote data, and encourages innovative uses and analyses of materials data.

We have already started to see new and novel uses of the MP data via the Materials API and the pymatgen library, such as screening for CO_2 sorbents, calculation of x-ray spectra for clusters of atoms, and performing Voronoi analysis to find possible interstitial sites.

IV. Discussion

We discuss here the distinct challenges that the Materials Project and similar efforts face as they scale up to large parallel environments. The discussion will be structured around the four main components of the architecture shown in Figure 2: parallel computation, data analytics, data validation and verification, and data dissemination.

Our experience highlights three future areas of research and exploration that are required: (1) how to provision and share (across projects) a set of resources to perform data analytics and loading between the parallel compute cluster and the database, (2) how to perform bi-directional sharing of both good data and good code to operate on it, and (3) how to automate and scale continuous validation and verification functions on dynamic datastores.

A. Parallel computation

This section discussions challenges encountered with running our workflows in parallel HPC environments, corresponding to the "Parallel computation" box in Figure 2.

1) Batch queue limitations: Most HPC systems allow only a handful of queued jobs per user and batch queueing systems like PBS are designed with this in mind. But for many of the high throughput workloads like the Materials Project, there are thousands of small jobs. In the MP, we worked with NERSC to get advanced reservations that temporarily suspended these limits. We also address these limits with *task farming*, where a single job in the queue runs multiple VASP calculations; task farming also smooths large wallclock variations (see §III-C).

2) State management: MPI processes typically read and write their inputs and outputs that live on a shared file system that is accessible to all the MPI tasks. In contrast, high throughput computing might have a variety of different modes of task state synchronization and managing input and output. In our case, we use MongoDB as the central datastore for the tasks. There are a number of challenges to running MongoDB type datastores on HPC systems. First, most HPC systems are configured such that the internal worker nodes are not allowed to communicate outside the system. Thus, we had to use a proxy to have our tasks communicate with the MongoDB Server.

Recent systems used in HPC systems provide a Non-Uniform Memory Access (NUMA) architecture. In this architecture, each processor has local memory that provides lower latency memory access. All memory is accessible from all processors but at a potentially higher latency and lower performance. Databases such as MongoDB, where a single multi-threaded process uses most of the system's memory, are atypical workloads for these systems . Using the `numactl` program, it is possible to interleave the allocated memory with a minimal impact to performance. But this trend is accelerating: power efficiencies of many simpler processors are, as predicted [2], leading to 100's and 1000's of processors per chip, each with their own local memory. This raises the question of provisioning the right resources for databases for many-task workloads at HPC centers.

B. Data analytics

This section discusses the challenges with performing data analytics on the simulation data, corresponding to the "Data analytics" box in Figure 2.

1) Scaling community involvement: The *pymatgen* Python library powers computations within MP, and it is a powerful and useful tool. The challenge going forward is to bring additions and improvements to the codebase while keeping the codebase "clean" in both the sense of quality algorithms and good programming style. Requiring unit tests helps, but more subtle issues of style and clarity are much harder to police. From the perspective of the scientists, the problem is that (a) it takes time and effort to go from something that works to something that is stylistically correct, with no immediate reward, and (b) Computer Science provides very little in the

way of concrete guidelines, or tools. The social solution to this problem is to use Github in the traditional way of accepting patches, with the acceptance to the library being the "carrot" that encourages better coding. But this raises the broader issue that in an era in which scientists are increasingly expected to program, there are not sufficient models for modern scientific programming practices, *e.g.* using OO design in Python.

2) Scaling data analytics using Hadoop: MapReduce and Hadoop have gained traction in the last few years for scalable parallel analytics. There are several ways that MP can leverage these capabilities while retaining the coherence of a single datastore. First, MongoDB provides extensions that support analysis using Hadoop, directly. However, to effectively use MongoDB and Hadoop together, it is necessary to understand the performance trade-offs. In earlier work [6], we show that Hadoop can be several times faster than the built-in MongoDB MapReduce framework.

For larger-scale analytics, this may not be a good solution as MongoDB is significantly slower than HDFS as a backend store for MapReduce jobs. In this case, efficiency can be gained by pre-staging the MongoDB data to HDFS. This would make sense, for example, for experimental image data from light sources. Even when HDFS is being used directly, MongoDB will continue to contain references to the data that allow queries to be performed using the QueryEngine abstraction layer described in §III-B4.

C. Data loading, validation, and verification

This section discusses the challenges with all the operations required between when the data is generated and analyzed and the dissemination of the data to the user, corresponding to the "Data V&V" box in Figure 2.

1) Data loading: The process of loading output data from the VASP simulation into the database is performed as a post-processing step. This is necessary because the "worker" nodes cannot connect out to the database server and, at any rate, this would be a poor use of optimized parallel resources. Currently, this is a manual operation that takes a significant time. We are working with NERSC to transition to a more automated, incremental loading capability that can run on utility resources. This is one instance of a broader need, *e.g.*, similar capabilities could help other projects to automate their post-processing steps, or help MP do backups and replication. We believe that this points to the necessity for computing centers such as NERSC to allocate manpower and system resources to a more scalable model for serving this need, and we hope that what we learn from automating the data loading and data mining in MP will inform that model.

2) Validation and verification: Continuous validation and verification (V&V) should be part of any infrastructure that centers on a datastore. Our current approaches to validation and verification of the database involve selected manual tests of known compounds, and some automated consistency tests. This is incomplete and has led to a number of last-minute scrambles to fix a calculation bug before releasing a database.

A logical language in which to write the V&V of a database is MapReduce, with the Map finding the items to compare and the Reduce performing the comparisons. MongoDB's built-in MapReduce functionality is severely limited by implementation within a single-threaded Javascript engine. MP currently uses a simple custom MapReduce framework written in Python. The Mongo/Hadoop connector has better performance, as described in Section IV-B2. As for the data loading problem discussed above, automated V&V would benefit from a dedicated infrastructure that is connected directly to the database.

D. Data sharing and dissemination

We discuss the challenges associated with security and privacy and query performance as related to data sharing and dissemination.

1) Security and Privacy: In creating a public data resource, it is critical to protect the data and its users from misuse. Challenges include maintaining privacy of user data and preventing malicious queries from crippling the system.

Rather than maintaining sensitive user login information, we delegate authentication to trusted third party providers (like Google or Yahoo). This also simplifies account management process since anyone with an email address from a trusted third party can sign up for an an account. Additionally, any data generated by the user using the MP tools can be made private or public.

Because all queries go through the *QueryEngine* abstraction layer described in §III-B4, all queries are sanitized and cannot access the database directly. We also implement checks to limit the number of queries from a given user to prevent denial-of-service or data scraping attacks. In the end we recognize that security is an ongoing concern, and we work with our systems administrators to follow best-practices across the board.

2) Query performance: Query performance is an important practical consideration for a database that must serve users across the world. Despite a growing number of users, the database has performed well for interactive searches originating from the web UI. The distribution of query times, across all collections, between April 24 to August 31, 2012 is shown in the histogram in Figure 5.

A majority of the queries are on the order of a few hundred milliseconds. The few outliers are still well within the range of user expectations for response time on a web portal. The scatterplot inset in Figure 5 shows a time-series of individual query times for the last half of August 2012, which is the most recent data as of this writing.

Our query performance has so far been good with only a single MongoDB server. Future scalability can leverage the sharding and replication capabilities built in to MongoDB [21]. This will allow us to maintain performance at scale as the Materials Project data grows, as well as isolate the various roles of the database to separate servers for performance reasons.

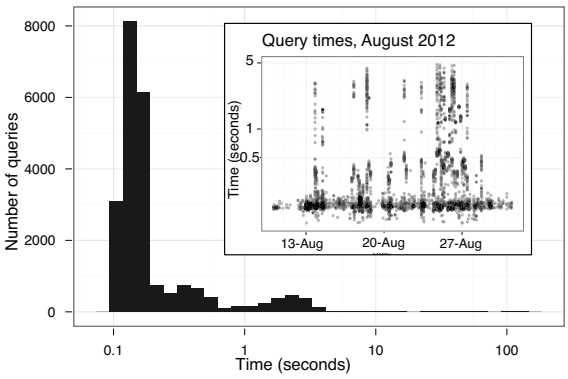

Fig. 5: Histogram of query performance from April 24 to August 31, 2012. Inset: time-series of query performance in August 2012.

V. Conclusion

In this paper, we described the Materials Project infrastructure that uses a NoSQL document store as the center of its infrastructure for meeting the challenges of high-throughput DFT calculations in HPC environments. This approach has several advantages in terms of flexibility, productivity, and scalability.

In addition, we identified several areas for future research that could improve the HPC ecosystem needed for these studies. Addressing these areas could have a signficant impact on progress for other scientific domains, as the Materials Project is an examplar of a building wave of data-centric sciences. We also highlight the importance of the data dissemination thrust of such projects, *e.g.* the Materials API and *pymatgen*, which has a large impact on the scientific community for using shared data.

Future work in the Materials Project will address the challenges associated with allowing users to define workflows on their own protected datastores. This will enable broader collaborative science by shortening the materials design cycle.

This work was supported by the Director, Office of Science, Office of Basic Energy Sciences, of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231

References

[1] G. Bergerhoff, R. Hundt, R. Sievers, and I. D. Brown. The inorganic crystal structure data base. *Journal of Chemical Information and Computer Sciences*, 23(2):66–69, 1983.

[2] K. A. R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, and K. A. Yelick. The landscape of parallel computing research: A view from berkeley. Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, Dec. 2006.

[3] R. Cattell. Scalable sql and nosql data stores. *SIGMOD Rec.*, 39(4):12–27, May 2011.

[4] G. Ceder, G. Hautier, A. Jain, and S. Ong. Recharging lithium battery research with first-principles methods. *MRS Bulletin*, 36(03):185–191, 2011.

[5] S. Curtarolo et al. AFLOW: An automatic framework for high-throughput materials discovery. *Computational Materials Science*, 58:218–226, June 2012.

[6] E. Dede, M. Govindaraju, D. Gunter, R. Canon, and L. Ramakrishnan. Semi-structured data analysis using mongodb and mapreduce: A performance evaluation. *In Submission*.

[7] E. Deelman, J. Blythe, Y. Gil, and C. Kesselman. Workflow Management in GriPhyN. Grid Resource Management, J. Nabrzyski, J. Schopf, and J. Weglarz editors, Kluwer, 2003.

[8] T. W. Eagar. Bringing new materials to market. *Technol. Rev.*, 98(2):42–49, Feb. 1995.

[9] ESG-NCAR Gateway. http://www.earthsystemgrid.org/.

[10] G. Hautier, A. Jain, H. Chen, C. Moore, S. P. Ong, and G. Ceder. Novel mixed polyanions lithium-ion battery cathode materials predicted by high-throughput ab initio computations. *J. Mater. Chem.*, 21:17147–17153, 2011.

[11] G. Hautier, A. Jain, and S. P. Ong. From the computer to the laboratory: materials discovery and design using first-principles calculations. *Journal of Materials Science*, 47(21):7317–7340, Nov. 2012.

[12] G. Hautier, A. Jain, S. P. Ong, B. Kang, C. Moore, R. Doe, and G. Ceder. Phosphates as lithium-ion battery cathodes: An evaluation based on high-throughput ab initio calculations. *Chemistry of Materials*, 23(15):3495–3508, 2011.

[13] J. S. Hummelshøj, F. Abild-Pedersen, F. Studt, T. Bligaard, and J. K. Nørskov. Catapp: A web application for surface chemistry and heterogeneous catalysis. *Angewandte Chemie International Edition*, 51(1):272–274, 2012.

[14] A. Jain, G. Hautier, C. J. Moore, S. P. Ong, C. C. Fischer, T. Mueller, K. A. Persson, and G. Ceder. A high-throughput infrastructure for density functional theory calculations. *Computational Materials Science*, 50(8):2295 – 2310, 2011.

[15] Introducing JSON. http://www.json.org/.

[16] Systems Biology Knowledgebase. http://kbase.us/.

[17] G. Kresse and J. Furthmüller. Efficient iterative schemes for ab initio total-energy calculations using a plane-wave basis set. *Phys. Rev. B*, 54:11169–11186, Oct 1996.

[18] Materials Genome Initiative for Global Competitiveness. http://www.whitehouse.gov/sites/default/files/microsites/ostp/materials_genome_initiative-final.pdf.

[19] Materials Project. http://www.materialsgenome.org/.

[20] Y. Mo, S. P. Ong, and G. Ceder. First principles study of the li10gep2s12 lithium super ionic conductor material. *Chemistry of Materials*, 24(1):15–17, 2012.

[21] A Brief Introduction to MongoDB. http://www.10gen.com/static/downloads/ mongodb_introduction.pdf.

[22] S. P. Ong, V. L. Chevrier, G. Hautier, A. Jain, C. Moore, S. Kim, X. Ma, and G. Ceder. Voltage, stability and diffusion barrier differences between sodium-ion and lithium-ion intercalation materials. *Energy Environ. Sci.*, 4:3680–3688, 2011.

[23] S. P. Ong, W. D. Richard, A. Jain, G. Hautier, M. Kocher, S. Cholia, D. Gunter, V. Chevrier, K. A. Persson, and G. Ceder. Python materials genomics (pymatgen) : A robust, open-source python library for materials analysis. *Computational Materials Science*, 2012 (Submitted).

[24] pymatgen: Python Materials Genomics library. http://packages.python.org/pymatgen/.

[25] Quantum materials informatics project. http://www.qmip.org/.

[26] I. Raicu and I. T. Foster. Many-task computing for grids and supercomputers. *2008 Workshop on ManyTask Computing on Grids and Supercomputers*, pages 1–11, 2008.

[27] R. Raman, M. Livny, and M. Solomon. Matchmaking: Distributed Resource Management for High Throughput Computing. In *In Proceedings of the Seventh IEEE International Symposium on High Performance Distributed Computing*, volume 7, pages 140–146. Citeseer, IEEE Comput. Soc, 1998.

[28] Sloan Digital Sky Survey: Mapping the Universe. http://www.sdss.org/.

[29] Vienna Ab-Initio Simulation Package. http://www.vasp.at/.

[30] M. Wilde, M. Hategan, J. M. Wozniak, B. Clifford, D. S. Katz, and I. Foster. Swift: A language for distributed parallel scripting. *Parallel Computing*, 37(9):633–652, 2011.

[31] D. N. Williams et al. The earth system grid: Enabling access to multi-model climate simulation data. *Bulletin of the American Meteorological Society*, 90(2):195–205, 2012/09/07 2009.