

Computational Materials Repository

User Manual

db_keywords	csv import AdsorbatesOnPureElements.csv	csv import AdsorbatesOnPureElements.csv	csv import AdsorbatesOnPureElements.csv	csv import AdsorbatesOnPureElements.csv	csv import AdsorbatesOnPureElements.csv
adsorbate	5	SH	CH	NH	NO
atomic_number	[16, 13]	[16, 1, 78]	[16, 1, 45]	[7, 1, 47]	[7, 8, 77]
authors	Abild-Pedersen F, Greeley J, Studt F, Rossmeisl J, Munter TR, Moses PG, Bligaard T, Norskov JK	Abild-Pedersen F, Greeley J, Studt F, Rossmeisl J, Munter TR, Moses PG, Bligaard T, Norskov JK	Abild-Pedersen F, Greeley J, Studt F, Rossmeisl J, Munter TR, Moses PG, Bligaard T, Norskov JK	Abild-Pedersen F, Greeley J, Studt F, Rossmeisl J, Munter TR, Moses PG, Bligaard T, Norskov JK	Munter TR
bindingenergy	-2.99292	12.908203	-6.1308594	-2.0332031	-1.6035156
crystalstructure	fcc	fcc	fcc	fcc	fcc
date	2007-11-08 16:01:11.348	2007-11-08 16:01:14.276	2007-11-08 16:00:49.168	2007-11-08 16:01:02.86	2007-11-08 16:01:07.812
db_calculator	spreadsheet	spreadsheet	spreadsheet	spreadsheet	spreadsheet
db_calculator_instance	443.444M	443.444M	443.444M	443.444M	443.444M
db_cmvr_schema	spreadsheet_03_20110523	spreadsheet_03_20110523	spreadsheet_03_20110523	spreadsheet_03_20110523	spreadsheet_03_20110523
db_description	Import of VMDF data	Import of VMDF data	Import of VMDF data	Import of VMDF data	Import of VMDF data
db_file_name	9771421_0162542131_000001.db	9771421_0162542131_000001.db	9771421_0162542131_000001.db	9771421_0162542131_000001.db	9771421_0162542131_000001.db
db_file_version	0.3	0.3	0.3	0.3	0.3
db_hash	006f4a73613487483872c6ac3c8dd61da97d6d2e	00934b4dd4c3d2f83b14a7ef9fe74bce95cc47	01d061886fab7a2c4499ee892a9ae5a64aeaa0	02a72f236df9a0	0304ce053c0263f80f0dce372c7f8e794
db_hash_schema	default	default	default	default	default
db_hash_type	sha1	sha1	sha1	sha1	sha1
db_hashed_columns	{db_calculator, db_calculator_instance, db_cmvr_revision, db_date, db_file_version, db_hash_type, db_hashed_columns, db_location, db_pid, db_user, db_xml_schema, adsorbate, atomic_number, authors, bindingenergy, crystalstructure, date, description, facet, id, layers, name, reference, referencesystem, site, slab, symmetry, title, totalenergy, type}	{db_calculator, db_calculator_instance, db_cmvr_revision, db_date, db_file_version, db_hash_type, db_hashed_columns, db_location, db_pid, db_user, db_xml_schema, adsorbate, atomic_number, authors, bindingenergy, crystalstructure, date, description, facet, id, layers, name, reference, referencesystem, site, slab, symmetry, title, totalenergy, type}	{db_calculator, db_calculator_instance, db_cmvr_revision, db_date, db_file_version, db_hash_type, db_hashed_columns, db_location, db_pid, db_user, db_xml_schema, adsorbate, atomic_number, authors, bindingenergy, crystalstructure, date, description, facet, id, layers, name, reference, referencesystem, site, slab, symmetry, title, totalenergy, type}	{db_calculator, db_calculator_instance, db_cmvr_revision, db_date, db_file_version, db_hash_type, db_hashed_columns, db_location, db_pid, db_user, db_xml_schema, adsorbate, atomic_number, authors, bindingenergy, crystalstructure, date, description, facet, id, layers, name, reference, referencesystem, site, slab, symmetry, title, totalenergy, type}	{db_calculator, db_calculator_instance, db_cmvr_revision, db_date, db_file_version, db_hash_type, db_hashed_columns, db_location, db_pid, db_user, db_xml_schema, adsorbate, atomic_number, authors, bindingenergy, crystalstructure, date, description, facet, id, layers, name, reference, referencesystem, site, slab, symmetry, title, totalenergy, type}
db_last_modified	2011-08-04 12:39:04.768543	2011-08-04 12:39:04.208608	2011-08-04 12:39:04.208608	2011-08-04 12:39:04.208608	2011-08-04 12:39:04.608887
db_location	CAMD Physics, DTU	CAMD Physics, DTU	CAMD Physics, DTU	CAMD Physics, DTU	CAMD Physics, DTU
db_pid	21877	21877	21877	21877	21877
db_xml_schema	spreadsheet_03_20110523.xsd	spreadsheet_03_20110523.xsd	spreadsheet_03_20110523.xsd	spreadsheet_03_20110523.xsd	spreadsheet_03_20110523.xsd
description	Binding energy for adsorbates on surfaces of pure elements	Binding energy for adsorbates on surfaces of pure elements	Binding energy for adsorbates on surfaces of pure elements	Binding energy for adsorbates on surfaces of pure elements	Binding energy for adsorbates on surfaces of pure elements
facet	[111]	[111]	[111]	[111]	[111]
id	79284	79284	79284	79284	79284
layers	3	10	3	3	3
name	S/Al	SH/Pt	CH/Rh	NH/Ag	NO/Ir
reference	PRL 99, 016105 (2007)	PRL 99, 016105 (2007)	PRL 99, 016105 (2007)	PRL 99, 016105 (2007)	Center for Atomic-scale Materials Design Technical University of Denmark, Denmark (2007)
referencesystem	78759	78848	78767	78758	78763
site	top	hcp	hcp	fcc	top
slab	Al	Pt	Rh	Ag	Ir
symmetry	p2x2	1x2	p2x2	p2x2	p2x2
title	Scaling properties of adsorption energies for hydrogen containing molecules on transition metal surfaces	Scaling properties of adsorption energies for hydrogen containing molecules on transition metal surfaces	Scaling properties of adsorption energies for hydrogen containing molecules on transition metal surfaces	Scaling properties of adsorption energies for hydrogen containing molecules on transition metal surfaces	Towards catalysis informatics (Ph.D thesis)

April 2012
rev. 1.0

Query: (Hint 1: use * to search for partial keywords; Hint 2: use " for keyword with spaces)

db_user=jyca columns="id_ref,db_date,db_calculator/instance,db_user,db_description,db_keywords,atoms,downloads,Epot"

Results per Page: 10

Execute or get result as ☐ % or ☐ json

Restrict keywords:

- mon + (6001)
- mon + (3008)
- ABO2N + (2039)
- ABO2N + (1982)
- ABO2N + (1980)
- ABO2N + (1542)
- perovskite + (1542)
- rutile + (1257)
- ABO4 + (1257)

Found 13237 results (0.015s)

id_ref	db_date	db_calculator/instance	db_user	db_description	db_keywords	atoms	downloads	Epot
50	2011-02-18 10:28:22	gspan	jyca	Screening cubic perovskite structures, transition metal oxynitride	ABO2N mon	Ir Ni O Ti Zn (1M, 10, 15)	1	-0.251749761691
55	2011-03-02 23:05:38	gspan	jyca	Screening cubic perovskite structures, transition metal nitride	ABO2N mon	Ir Ni O Ti Zn (1M, 10, 15)	1	-0.421213452195
63	2011-04-15 16:59:23	gspan	jyca		ABO2N mon	Ir Ni O Ti Zn (1M, 10, 15)	1	-0.421001268839
136	2011-04-15 17:04:49	gspan	jyca		ABO2N mon	Ir Ni O Ti Zn (1M, 10, 15)	1	-0.351776539882
120	2011-03-02 23:30:28	gspan	jyca	Screening cubic perovskite structures, transition metal nitride	ABO2N mon	Ir Ni O Ti Zn (1M, 10, 15)	1	-0.492712087712
122	2011-02-18 10:06:57	gspan	jyca	Screening cubic perovskite structures, transition metal oxynitride	ABO2N mon	Ir Ni O Ti Zn (1M, 10, 15)	1	0.432193505699
138	2011-03-10 11:21:51	gspan	jyca	Screening cubic perovskite structures, transition metal oxynitride	ABO2N mon	Ir Ni O Ti Zn (1M, 10, 15)	1	-1.65142424208
143	2011-04-15 16:33:24	gspan	jyca		ABO2N mon	Ir Ni O Ti Zn (1M, 10, 15)	1	-0.917666547933
151	2011-02-18 10:01:25	gspan	jyca	Screening cubic perovskite structures, transition metal oxynitride	ABO2N mon	Ir Ni O Ti Zn (1M, 10, 15)	1	0.176565558374
158	2010-12-14 16:16:51	gspan	jyca	Screening cubic perovskite structures, transition metal oxides	ABO2N mon	Ir Ni O Ti Zn (1M, 10, 15)	1	-0.405471998093

Document History

This document bases on the thesis of David D. Landis “Computational Materials Repository” from 2012, CAMD, DTU.

Modification History

- rev 1.0 April, 2012

Acronyms, Terms and Definitions

term	short explanation	chapter
add/commit/ submit/upload	the process of adding db-files to the db-file repository; from the db-file repository the data is uploaded to the MySQL database	3.3.4.1
agent	autonomous process for data analysis or tasks in the background	2.2, 2.9.3, 3.3.3.1
calculation	a short name for an <i>electronic structure calculation</i> performed by a code like GPAW or VASP	
code	<i>electronic structure simulator</i> e.g. GPAW or VASP	
CLI	the command line interface allows to run common tasks like showing content of db-files to be executed in a linux shell	2.5,3.3.1.1
CMR database	<i>see</i> database	
CMR schema	<i>see</i> schema	
cluster	a computer cluster or super computer that executes jobs (scripts)	
database	with database the MySQL database with the CMR database schema is meant	3.3.2.3
commit	<i>see</i> add	
db-file (cmr-files)	db-file is the internal file format of CMR; in the future these files will be called cmr-files to avoid naming conflicts with other file formats	3.3.3.2
db-file repository	the db-file repository stores db-files that are later uploaded to the MySQL database	3.3.2.1
field(s)	a field is a name/value pair e.g. energy/-0.1	2.1

group	a group contains a collection of results (calculations); it is also possible that groups contain other groups	2.2, 2.7
keyword(s)	keywords are used to identify calculations and can be added to all the data uploaded to the database	2.1
mapping	the process of renaming a field or changing the unit of the value dynamically when reading a db-file	2.9.2
MySQL database	<i>see</i> CMR database	
PHP/HTML (user) interface	a user interface of CMR that runs on a web server and is accessed with a web browser	2.3, 3.3.1.3
PHPUI	<i>see</i> PHP/HTML (user) interface	
PUI	<i>see</i> python (user) interface	
python (user) interface	a user interface of CMR	3.3.1.2
schema	the database/ CMR schema	3.3.3.3, 3.3.2.3
submit	<i>see</i> add	
SiGUI	a plug-in for Silo	3.3.1.4.1
Silo	a user interface that connects to the CMR database	3.3.1.4
unique identifier	a unique identifier is created for every db-file; this identifier is used to identify group members	2.6
UI	<i>see</i> unique identifier	
upload	<i>see</i> add	

Contents

1	Introduction	11
2	Introduction and usage of CMR	17
2.1	Working with CMR	18
2.2	Step by Step	20
2.3	PHP/HTML Web Interface	27
2.4	Querying and Analysis	34
2.5	The Command Line Interface	38
2.6	Modifying Data in the Database	40
2.7	Groups	43
2.8	What is actually stored in a db-file?	44
2.9	Advanced Task	47
2.9.1	Publishing results with CMR	47
2.9.2	Using a Mapping	47
2.9.3	Supporting a new file format	47
2.9.3.1	Step 1: create a converter	47
2.9.3.2	Step 2: create a CMR schema	49
2.9.3.3	Step 3: declare the plug-ins	49
2.9.3.4	Step 4: set environment variables	51
2.9.3.5	Step 5: create an XML schema	51
3	Computational Materials Repository	55
3.1	Overview	55
3.2	CMR Challenges	56
3.3	System Components and Processes	57
3.3.1	User Interfaces	59
3.3.1.1	Command line interface (CLI)	59
3.3.1.2	Python User Interface (PUI)	60
3.3.1.3	PHP/HTML User Interface (PHPUI)	61
3.3.1.4	Silo Framework, and the SiGUI plug-in	62
3.3.1.4.1	SiGUI, a silo plug-in	63
3.3.2	Repositories	64
3.3.2.1	db-file repository	64
3.3.2.2	local repository	65

3.3.2.3	CMR Database	65
3.3.3	Other Components	67
3.3.3.1	Agents	67
3.3.3.2	Db-files/cmr-files	68
3.3.3.3	Schemas	71
3.3.4	Processes	71
3.3.4.1	Conversion to /writing of a db-file	71
3.3.4.2	Upload	73
3.3.4.3	Validation	74
3.4	Tools	74
3.4.1	Motivation	74
3.4.2	Python	74
3.4.3	MySQL	75
3.4.4	Apache/PHP	76
3.4.5	Tools Summary	76
3.5	Outlook	77
3.5.1	Improvements of upload and validation	77
3.5.2	Improvements of the PHP/HTML user interface	77
3.5.3	A submission tool	77
3.5.4	CMR in the Cloud	78
3.6	Conclusion	78
4	Appendix	79
4.1	Extracted Values from Codes	79
4.2	PHPUI script to continue analysis in the PUI	82
4.3	Deployment Examples and Dependencies	84
4.3.1	Minimal	84
4.3.2	Medium	85
4.3.3	Institute or high quality database	85
4.4	Inside a db-file	86

Chapter 1

Introduction

The design of novel and versatile materials is of great importance for our society. This is reflected by the strong focus on discovering new materials for energy conversion and storage to provide a sustainable alternative to the fossil-based fuel economy. Atomic-scale calculations are becoming increasingly important in strengthening our ability towards meeting this challenge, as it over time has provided an ever-improving alternative to expensive experiments. As computational resources become more readily available and the methods more efficient, studies can be performed on complex systems. This poses a challenge in terms of systematic storage, retrieval and analysis of the results. If there is no automated process or predefined schema for the collection and analysis, every researcher will spend more and more time on administrating his/her data instead of deriving useful information. Other time consuming tasks that have to be done repeatedly are sharing intermediate results with collaborators, archiving data and documentation how it was gathered and making it available on the Internet.

Institutes and research facilities have a particular interest in aggregating and storing data in a uniform way: if done right they not only have access to former research data, but can optimize future calculations by finding better initial guesses of atomic positions, data mine older data or continue a project seamlessly.

In biological research, scientists realized early that sharing of the data, through commonly available databases improves research and leads to faster progress in the field. The National Center for Biotechnical Information (NCBI)[1] has the biggest and the most cited database in the molecular biology and medical field - and its free. The database was initiated in 1988 by the American government and consists of subsets of smaller databases that hold information about gene and protein structure, genetic variants, common domains, et cetera. Additionally NCBI links all published articles to relating gene/protein datasets. By typing the name of a gene in the search box, NCBI searches all its databases and provides cross-linked results to relating data sets which enables further investigations and analyses. Researchers are also able to upload their own data. Nowadays NCBI

is an integral part in daily routine of biologist researchers.

Other biological databases are Ensemble[2], which stores information about the genome, or the Protein Data Bank (PDB)[3] - which contain published 3D structures of proteins; a good example of collaborate efforts is the Human Genome Project[4] that was initiated in 1990 and aimed at sequencing the whole human genome. Many institutes around the world were participating in sequencing, analyzing, annotating and submitting data.

The electronic structure calculation community is not there yet. Especially we are far from a cross-linked freely available database where researchers can upload their data and put it in context with already available data in order to derive new results - however partial solutions exist: The Virtual Materials Design Framework (VMDF)[5], a tool to filter and analyze aggregated sets of electronic structure data introduces a first step towards user-friendly analysis of data, the CAMd database[6] or AFlowLib[7] that present a selection of aggregated data on the Internet and make it available through a web interface - or ICSD, the Inorganic Crystal Structure Database[8, 9] that collects validated experimentally determined data, and provides a software tool and a web interface to perform searches - unfortunately not for free. During the last eight months more elaborate tools and databases have seen the light of day: ESTEST[10] aims at validation and verification of electronic structure calculations of different codes; Materials Genome[11, 12] has a high-throughput infrastructure to perform screenings and presents results through toolboxes that are accessible on the Internet - and the Quixote[13, 14] project that focuses on quantum chemistry data presents a collaborative open source framework to collect and process data from different sources.

The reasons why we are not there is that it is hard to find a common applicable scheme that is able to cover most aspects. The electronic structure data is for instance more heterogeneous - there are numerous file formats - and the the analysis depends greatly on the perused goal. The challenges of handling data in an efficient and reusable way can be divided in a few mature tasks as represented in Fig. 1.1.

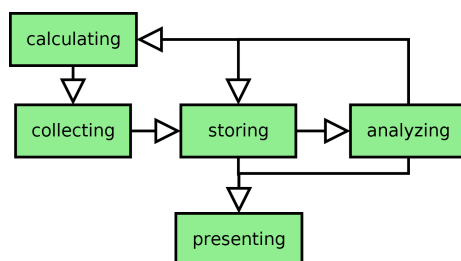


Figure 1.1: Major tasks when aggregating data - the arrows represent the work flow. **Calculating** produces results that need to be **collected** in a commonly readable format. These results could then be **stored** in a database for further **analysis**. The **presentation** can be done in a web-interface or any other tool that is able to retrieve data from the storage. Important is to notice that analysis can produce derived data that should be added to the storage as well.

With the Computational Materials Repository (CMR) that is presented in this document, we provide a modular open source system that addresses the challenges of collecting, storing, making data available through interfaces and propose and implement basic ideas how to perform analysis. The focus is in particular on DFT codes because they represent a favorable trade-off between speed and accuracy for the treatment of “few-hundred-atom” systems highly relevant for understanding physical and chemical properties of materials. We propose a file-format (db-files) that holds extracted data syntactically and semantically as close as possible to the original output file - and introduce groups to indicate connections between data. The Python user interface allows to search and query data which then can be analysed. In order to cope with different assumptions on names and units, CMR implements mappings. A mapping enables to rename and convert for example the field `CartesianPositions` in Bohr units, to `positions` in Angstrom.

To get a user-friendly way of looking at the data, the PHP/HTML interface presents the data with the help of a MySQL database and a webserver as shown in Fig. 1.2. When the data is published it is possible to create a custom view as shown in Fig. 1.3 to make the results easier accessible for public.

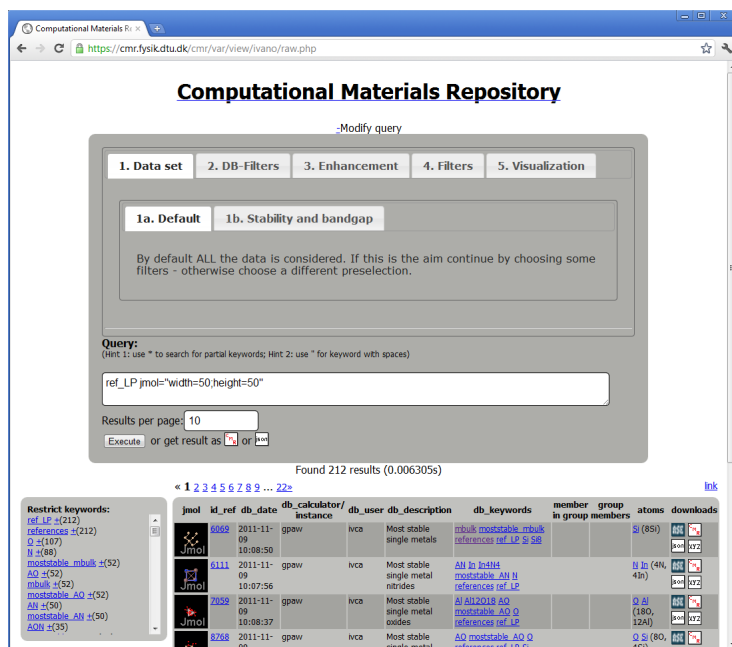


Figure 1.2: The PHP/HTML interface finds data based on keywords and fields, suggests related keywords and provides ways to download the data.

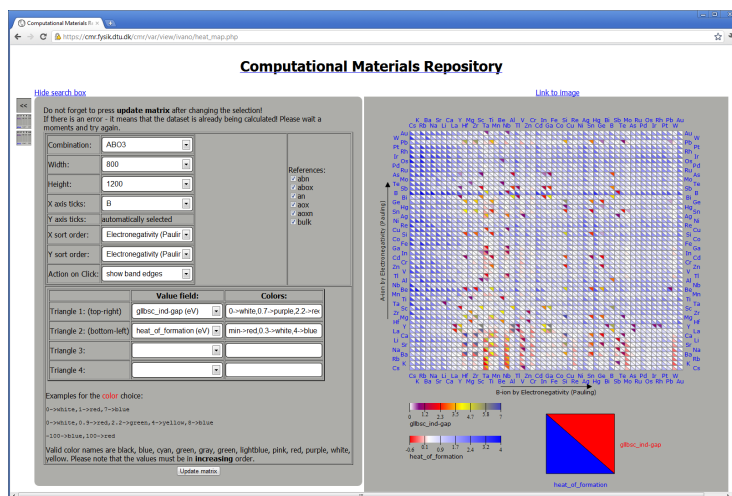


Figure 1.3: A customized view of the PHP/HTML interface to present published data in a heat-map.

The focus for the development of CMR was lied on generality, modularity and extensibility. This leaves a lot of room for performance tuning like using a

compiled language to verify the db-files or only uploading data to the database that is actually needed.

The Computational Materials Repository is developed in collaboration between J. Greeley from the Argonne National Laboratory and S. Nestorov from the University of Chicago and J. S. Hummelshøj, T. Bligaard, J. K. Nørskov from SUNCAT Center for Interface Science and Catalysis in Stanford and us - Center for Atomic-scale Materials Design. CMR is part of Quantum Materials Informatics Project[15], which aims at establishing the core technology for integrated computational materials design.

This documentation begins with an introduction by example to present the capabilities and usage of CMR followed by a chapter that provides details about the challenges, the work flows and technical information.

Chapter 2

Introduction and usage of CMR

This chapter shows what the Computational Materials Repository (CMR) can do for you with concrete examples and explanations of the user interfaces. The subsequent chapter focuses on concepts and the inner workings.

We shall first briefly give an overview of all the functionalities of CMR and then move on to more concrete illustrations of its usage.

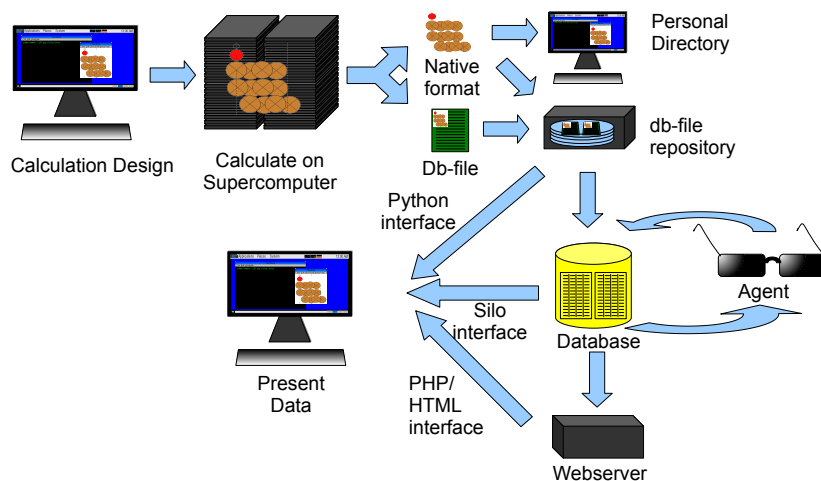


Figure 2.1: Overview over the infrastructure and the workflow of the data in CMR.

All the different components of CMR are depicted in Fig. 2.1. CMR simplifies the handling of heterogeneous data from electronic structure codes. It provides the same interface for handling the data independent of the original file format (and file names). The general interface makes scripting easy and reusable. Users of electronic structure codes are typically interested in obtaining and saving quantities like atomic positions, energies, and forces. Often though also code specific input parameters and results are relevant. For this reason CMR extracts most of the variables from the original output file and makes use of the ASE[16] (Atomic Simulation Environment) interface (if applicable) to get the relevant data in a unified representation in terms of variable names and units.

The extracted data is stored in a file format that we call **db-file**. db-files can be read on any linux machine that has Python[17] and CMR installed. The **command line interface** enables simple everyday tasks on these files like viewing content and performing basic editing. Normally all db-files are moved to a single directory that we call the **db-file repository**(3.3.2.1). With the CMR **python interface** queries can be run on the database as shown in section 2.2.

The data cannot only be stored in a collection of files. As depicted in figure 2.1 the data from the db-files can be uploaded to a database and queried with the python interface which will result in faster searches, because the data is indexed. More elaborate installations can make use of the **PHP/HTML interface** that provides a graphical user interface for searching and viewing data while **Silo**¹ features a workbench to analyze data. Last but not least so-called agents may run in the background – either invisible from the users or under user control – and perform certain tasks as for example grouping of db-files or preparing data needed by the user interfaces.

Currently CMR supports the import of GPAW, Dacapo, VASP and ASE trajectory files. Additionally CSV[18] files are supported which can be read and written by OpenOffice or Microsoft Excel.

2.1 Working with CMR

Working with a database is quite different from the common approach of storing data in a file system: files and directories are non-existent and therefore the data has to be identified in an other way. CMR implements an abstraction layer that simplifies finding of results by **keywords** or **fields**. A field is simply a name/value pair as for example energy/12.0 or program/gpaw.

In this section we discuss how to organize the data to be able to do the same as with the “old” approach and additionally profit from storing it with CMR in a database. Please note that the **python interface** supports to querying the **db-file repository** as well as the **database** (MySQL database) while **agents** work only with the database. However this does not change the way the data is organized.

¹**Silo** was initiated and is maintained by Jens Strabo Hummelshøj, at the time of writing a post-doc at SUNCAT Center for Interface Science and Catalysis at Stanford

There are few requirements that need to be identified in order to work efficiently with the database: First, the results need to be found after putting them into the database. Keywords replace directory and file names. For instance the file with the path `211surfaces/Ag211/edge/H.gpw` would get the keywords *211surface,Ag211,edge*. There is no need to put *H* as a keyword, for finding purposes, because we can search/restrict by atom type. However, if the count of the atoms matters as for N2 then it would make sense to include it. (The reason is that when looking for N2 we would also get the results with the single atom N.) The advantage of keywords over directory names is that the order doesn't matter and they enable to look at arbitrary subsets. When the two keywords *211surface* and *edge* and the requirement of presence of the atom H are combined in a query, we find all kinds of 211 surfaces with an H atom at the edge position. To look for these files in subdirectories would be considerably more cumbersome. Second, we would like to read the data in a similar way as the native output file. This is achieved with the python user interface as described later in section 2.4 or with the interface that views the data in a web browser as shown already in the introduction in Fig. 1.2. Third we profit from the database capability to search efficiently for the defined keywords and fields. Fourth, the results should provide more information than just the numbers from the output file. db-files are capable of storing scripts, in-/or output files, calculation parameters and custom fields as for example `surface=211`. Fifth, the results should be traceable. By creating groups (described later in the section 2.7) that reference the used data is conserved.

One might wonder what is the difference between the keyword *211surface* and the field `surface=211`, because both define the same data set. The different characteristics can be seen when grouping calculations according to criteria specific criteria. The chemisorption energy E_{chem} is calculated as $E_{chem} = E_Z - E_X - E_Y$, where E_X the adsorbate X in the gas-phase, E_Y the clean surface with atoms Y and E_Z is the total energy of surface with atoms Y with the adsorbate Z. If we had only the keywords, then we would have to know every single keyword and loop manually over all possible combinations. In pseudo code it would look as follows:

```
for surface in [111,211,...]:
    for adsorbate in [H, O, ...]:
        find result with keyword surface+adsorbate
        E_chem = ...
```

This is not efficient because every possible combination has to be checked - even if there is no data in the database and because the actually available surfaces and adsorbates have to be known. They cannot automatically be determined from the database.

A better way is to use the keyword to identify a certain type of calculation and the fields to combine them. This can be written as the following list of rules:

- X.keywords *contains* adsorbate
- Y.keywords *contains* surface

- Z.keywords *contains* surface+adsorbate
- X.ads=Z.ads
- Y.surf=Z.surf

These rules can then be translated to MySQL, the query language for the database. This approach is more general since the surfaces and adsorbates are determined automatically; the only thing that needs to be specified is the name of the fields (*ads* for the adsorbate and *surf*, for the surface). CMR provides a tool called GroupingAgent that translates the list of rules into query language. Its usage is explained in the next section 2.2.

To summarize: keywords *identify* a calculations or a certain type of calculation, while fields enable efficient *combination/grouping*. Groups show a *connection* between individual calculations or explain what calculations were involved to derive a result. Generally one would add a keyword to identify the type of calculation and fields for the properties that are needed to make the groups or to store additional information about the calculation.

2.2 Step by Step

In this section more concrete examples of how to work with CMR are presented. In order to actually run the presented python code, the user needs to have CMR installed – or have access to an institute that has deployed it. The CMR wiki and the installation instructions are available from <https://wiki.fysik.dtu.dk/cmr>.

The example takes its offset in a data set acquired as a computational screening for materials which can use solar light to split water - one potential way of producing hydrogen fuel. How the calculations are performed is described in detail in paper [19] “Computational Screening of Perovskite Metal Oxides for Optimal Solar Light Capture”. In the context here the following background should be sufficient.

For a material to be able to split water based on solar light, a number of conditions have to be fulfilled. First of all the material of course has to be stable also when surrounded by water. Furthermore, the electronic energy gap has to be in the right range (1.5-3 eV) so that the generated electron-hole pairs have sufficient energy to perform the water splitting (for this a large gap is preferred), but at the same time the gap should be small to increase the amount of light absorbed. There are some further conditions related to the positions of the band edges relative to the redox potentials for water splitting but we do not need to go into that here. The class of materials which have been investigated are mainly oxides (and oxynitrides) in the cubic perovskite structure. The interested reader is referred to more details in Paper [19] at the end of this thesis.

Here we focus on oxides in the perovskite structure which has the composition ABO_3 with A and B being metals. The above mentioned stability is investigated by defining the *heat of formation* (hof) E_{hof} of the ABO_3 compound as

$$E_{hof} = E_{ABO_3} - E_A - E_B - 3E_O$$

and we will show how this can be retrieved from the database as a list and also export it as a csv-file[18] (Comma Separated Values). To get an overview, the results should be displayed in a heat map with the PHP/HTML interface as shown in figure 2.2.

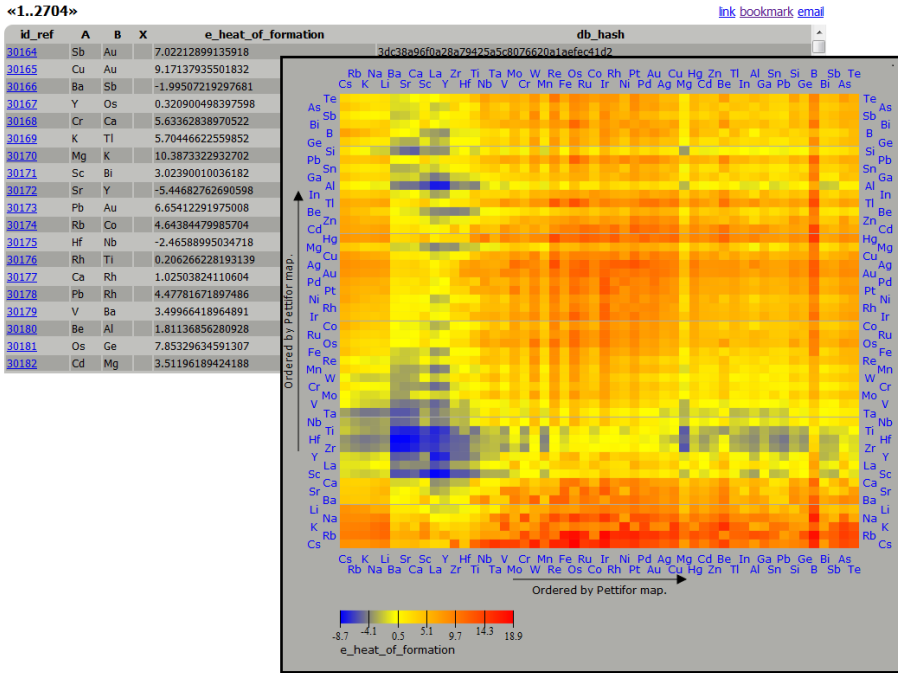


Figure 2.2: A table showing the numeric results and a heatmaps with the heat of formation for ABO₃. The elements are ordered along the axis according to Pettifor's stringing of the Periodic Table.

However, to begin with the beginning we should first get some data into CMR. The perovskite calculations have all been done with GPAW and this program is particularly well integrated with CMR so data transfer is easy. GPAW allows the use of the native `GPAW.write` function to directly produce a db-file as shown in figure 2.3. This has the advantage that the result is read directly from memory. Furthermore, at CAMD the db-file repository is a directory from where the data is automatically uploaded to the database, but CMR can also be used directly with the collection of db-files.

As is fairly obvious from the file, the format allows for the definition of a number of keywords ("ABO₃" and "CsZnO₃") as well as script-files and possibly other files that should be saved with this entry. It is also possible to define

```

import cmr ...
calc = GPAW() ...
cmr_params = {
    "db_keywords":["ABO3", "CsZnO3"],
    "db_scripts":["CsZnO3.py"],
    "db_files":["CsZnO3.txt"],
    "name":"CsZnO3",
    "A":"Cs",
    "B":"Zn",
    "X":"O3"
}
calc.write(".db", cmr_params=cmr_params)

```

Figure 2.3: A small python script to add an arbitrary result to the db-file repository. GPAW provides the ability to use its native write function.

new so-called *fields* which are name–value pairs. The last field in the example for example has the name “X” with the value “O3”. Some of the information carried by the fields or keywords are in this example somehow redundant. It is of course possible just from the identity of the atoms and their positions to recover which metal atom is in fact the “A” atom in the perovskite structure, but it may be convenient sometimes to add the information explicitly for fast and easy retrieval. If other programs than GPAW are used or if one wants to “manually” add information to the db-files this can be done as shown in figure 2.4.

```

import cmr

cmr_params = {
    "input":"CsZnO3.xxx",
    "output":".db",
    "db_keywords":["ABO3", "CsZnO3", "hof"],
    "db_scripts":["CsZnO3.py"],
    "db_files":["CsZnO3.txt"],
    "name":"CsZnKO3",
    "A":"Cs",
    "B":"Zn",
    "X":"O3"
}
cmr.convert(cmr_params)

```

Figure 2.4: A small python script to add an arbitrary result to the db-file repository.

All extra fields and keywords are defined in a python dictionary and passed as an extra argument to `cmr.convert`. “input” specifies the file that should be imported and “output” the name of the output file - using “.db” as a name signals that we are not interested in a real filename and would like the output to be written directly to the db-file repository (from where it will be added to the database). “db_keywords” is a list of keywords that identify this calculation, db_scripts and db_files take a list of filenames that will be included - and finally

"A", "B" and "X" are user defined fields.

The calculation of the heats of formation require input from several different GPAW calculations. The link between different calculations in CMR is established by creating so-called *groups* of calculations: The resulting calculated energy, $E_{hof,CsZnO_3}$, will be linked to a group containing the calculations for $CsZnO_3$, Cs, Zn, and O. When loading the data into CMR they have been equipped with keywords and fields as illustrated in figure 2.3, so the A- and B-metal atoms in a given perovskite calculation can be referred to using the fields "A" and "B". The keywords make finding a result easier - there are no file and directory names in a database that provide hints. When choosing keywords and fields, it is important to assure that results can be identified uniquely. The above proposed fields are sufficient for the presented example, but if there would be reference calculations for AO_2 , they could not be distinguished from the calculations of A: looking for the keyword "reference" and $A=Cu$ would return both of the results. In that case one would have to either add another keyword (e.g. bulk, oxide) or add a new field (e.g. type="bulk", type="oxide") to distinguish them. Alternatively, the atoms entering the calculation could be directly analyzed.

The easiest way to visualize the relevant grouping, is to use the PHP/HTML interface which provides an intuitive way to write queries. A screen shot is shown in figure 2.5 on page 24. The query that was used is **ABO3 A=Cs** and shows all results having the keyword "ABO3" and containing a field "A" with the value "Cs". The rest of the query **columns=...** in the figure selects the columns and defines to display the atoms, that should be displayed. These argument are optional, but convenient when viewing a data set with custom fields.

Now we would like to calculate the heat of formation (hof) by finding matching calculations. To make the syntax easier we denote in the following ABO_3 as the calculation of ABO_3 and $ABO_3.PotentialEnergy$ as the potential energy of ABO_3 . The hof is evaluated as

$$E_{hof} = ABO_3.PotentialEnergy - A.PotentialEnergy/A.nA \\ - B.PotentialEnergy/B.nA - X.PotentialEnergy$$

where $A.nA$ denotes the number of atoms of type A. We therefore need to find the four calculations in order to perform the computation.

The combination criteria can be written as three equations and be passed on to the *GroupingAgent*.

- $ABO_3.A = A.A$
- $ABO_3.B = B.A$
- $ABO_3.X = X.A$

The items A, B, X and ABO_3 are identified by the keywords. We also include the user name (db_user=ivca) to prevent confusions - other users might have used the same keywords and fields for other purposes and we don't want to mix results:

- A, B and X are identified with
 - keyword: reference
 - name-value: db_user=ivca

Computational Materials Repository

[Modify query](#)

1. Data set
2. DB-Filters
3. Enhancement
4. Filters
5. Visualization

1a. Default

By default ALL the data is considered. If this is the aim continue by choosing some filters - otherwise choose a different preselection.

Query:
(Hint 1: use * to search for partial keywords; Hint 2: use " for keyword with spaces)

ABO3 A=Cs columns=id_ref,A,B,anion,ase_potential_energy,atoms,downloads jmol="width=50;height=50"

Results per page:

or get result as or

Found 52 results (0.811766s) [link](#)

Restrict keywords:

- [Cs ±\(52\)](#)
- [O3 ±\(52\)](#)
- [ABO3 ±\(52\)](#)
- [perovskite ±\(52\)](#)
- [Q ±\(52\)](#)
- [mox ±\(52\)](#)
- [B ±\(1\)](#)
- [Rb ±\(1\)](#)
- [Na ±\(1\)](#)
- [V ±\(1\)](#)
- [Ba ±\(1\)](#)

















jmol	id_ref	A	B	anion	ase_potential_energy	atoms	downloads
	2147	Cs	Nb	O3	-34.4162	Cs Nb O (1Cs, 1Nb, 3O)	  
	2204	Cs	Bi	O3	-22.7017	Bi Cs O (1Bi, 1Cs, 3O)	  
	2207	Cs	B	O3	-21.8500	B Cs O (1B, 1Cs, 3O)	  
	2278	Cs	Be	O3	-20.1440	Be Cs O (1Be, 1Cs, 3O)	  

Figure 2.5: The query "ABO3 A=Cs" in the PHP/HTML interface and its result. `columns=` selects the columns that should be shown.

ABO₃ is identified with

- keyword: ABO3
- name-value: db_user=ivca

The **GroupingAgent** is a tool in the python interface that helps to group calculations. Normally agents are used to perform expensive queries and tasks in the background, but we can also make use of them directly. In our case we use one to calculate the heat of formation. The complete script that is being discussed here is shown in figure 2.6.

```
from cmr.ui.db_reader import DBReader
from cmr.plugins.agents.grouping_agent import GroupingAgent
hartree = 27.211395655517308

ABO3 = {"name": "ABO3",
        "name_value": {"db_user": "ivca"},
        "keywords": ["ABO3"],
        "fields_of_interest": ["PotentialEnergy as e_ABO3", "A", "B"]}
A = {"name": "A",
     "name_value": {"db_user": "ivca"},
     "keywords": ["references"],
     "fields_of_interest": ["PotentialEnergy as e_A"]}
B = {"name": "B",
     "name_value": {"db_user": "ivca"},
     "keywords": ["references"],
     "fields_of_interest": ["PotentialEnergy as e_B"]}
X = {"name": "X",
     "name_value": {"db_user": "ivca"},
     "keywords": ["references"],
     "fields_of_interest": ["PotentialEnergy as e_X"]}

comb_crit = [("ABO3.A", "A.A"), ("ABO3.B", "B.A"), ("ABO3.X", "X.A")]

calc = "ABO3.PotentialEnergy - ABO3.nA*A.PotentialEnergy/A.nA - ";
calc += "ABO3.nB*B.PotentialEnergy/B.nA - X.PotentialEnergy"

evaluations = {"e-heat_of_formation": "(%s)*%f" % (calc, hartree)}

keywords = ["ABO3_hof"]

items = [ABO3, A, B, X]

combination_criteria = comb_crit

agent = GroupingAgent(DBReader(),
                      items,
                      combination_criteria,
                      evaluations,
                      db_files=("", "members_only",
                                {"db_keywords": keywords}))

agent.run()
```

Figure 2.6: The calculation of the heat of formation with use of the **GroupingAgent**. (Section 2.2)

At the beginning, right after the imports, the definitions describe the data sets (ABO₃, A, B, X) that we need to retrieve from the database and the fields that we would like to see in the output (fields_of_interest):

```
ABO3 = {"name": "ABO3",
        "name_value": {"db_user": "ivca"},
        "keywords": ["ABO3"],
        "fields_of_interest": ["PotentialEnergy as e-ABO3",
                                "A",
                                "B"]}
...
```

The **name_value** (db_user=ivca) and the **keywords** (ABO₃) define how ABO₃ calculations are identified. The **fields_of_interest** contains information about fields that we would like to see in the output while the **name.field name** is used in the definition of the combination criterion:

```
comb_crit = [("ABO3.A", "A.A"),
              ("ABO3.B", "B.A"),
              ("ABO3.X", "X.A")]
```

For every quadruple that fits the criteria, we determine the hof by writing it as an equation:

```
calc = "ABO3.PotentialEnergy"
calc += "- ABO3.nA*A.PotentialEnergy/A.nA"
calc += "- ABO3.nB*B.PotentialEnergy/B.nA"
calc += "- X.PotentialEnergy"

evaluations = {"e-heat_of_formation": "(%s)*%f" % (calc, hartree)}

keywords = ["ABO3_hof"]
```

The last step is to add some keywords to identify the results and create an instance of the **GroupingAgent** and call its run method. During the run the created db-files (results) are written directly to the db-file repository.

```
agent = GroupingAgent(DBReader(),
                      [ABO3, A, B, X],
                      combination_criteria,
                      evaluations,
                      db_files=(" ", "members_only",
                                {"db_keywords": keywords}))

agent.run()
```

Instead of writing the data back to the database we can print it as well onto the screen and create a spreadsheet in the csv format. The following slightly modified code shows how:

```
...
dc = DataCollection()
agent = GroupingAgent(DBReader(),
                      criteria,
                      combination_criteria,
                      evaluations,
                      data_collection = dc)
```

```
agent.run()

dc.print_table(-1, columns=["A", "B", "X", "e_heat_of_formation"])
open("all.csv", "w").write(dc.get_csv())
```

After running the agent, the `DataCollection` object contains the result. Calling `print_table` prints a table and `get_csv` returns a string in the csv format that can be written to disk. A further going introduction to the `DataCollection` will be given in the next section 2.4.

The earlier created hof results can be found with the keyword `ABO3_hof`. A click on the id in the PHP/HTML user interface shows the `e_heat_of_formation` along with with links to the calculations that were used to obtain the number. In order to create the heat map shown in figure 2.2 we open the heat map tab and choose the parameters as shown in figure 2.14 and click the "Add" button which will add the selection to the query. After pressing "Execute" the heatmap is shown.

2.3 PHP/HTML Web Interface

The PHP/HTML web interface was already mentioned, but the tool and its usage shall now be discussed beginning with the creation of a query and followed by a description of how to use navigate on the result page.

Query Creation

The whole web page is divided into the query box and the result view below as already shown in Fig. 2.5. The **query box** (Fig. 2.7) is again divided into two parts: the lower with a text box where a query can be written and the upper that actually helps to create the query. The tabs in the top row are ordered according to the expected workflow: First the data sets are selected, then the fields and keywords and atoms are restricted. As a next step the retrieved data can be enhanced which means something is added to the result. This could be for example a link or some simple mathematical operation on fields. Then filter can be applied again. The difference between the db-filters and the filters are that the db-Filters are executed by the database (which is very fast) and the filters are applied after the retrieval (slow). The reason for having these filters is that we might want to perform operations on a calculated field. As a last step we can decide how to visualize the result.

In the above screenshot two data sets can be chosen: the default one which is simply all data in the database and the stability and band gap screening which limits the data set to this specific purpose. For now we choose the default which means we simply go to tab db-filter. Again we get a few sub-tabs presented: the first options to restrict by the user, by the type of calculation and by keywords:

In order to select a user we open the dropdown box that is labeled with "User" and choose a name. When done, we press "Add to query", which will add in our case the text `db_user=martebjo` to the query. We could press "Execute"

The screenshot shows the '1. Data set' tab of the CMR interface. It contains two sub-tabs: '1a. Default' and '1b. Stability and bandgap'. The '1a. Default' sub-tab is active, displaying the text: 'By default ALL the data is considered. If this is the aim continue by choosing some filters - otherwise choose a different preselection.' Below this, there is a 'Query:' section with a hint: '(Hint 1: use * to search for partial keywords; Hint 2: use " for keyword with spaces)'. A large text input field is provided for the query. At the bottom, there is a 'Results per page:' dropdown set to '10', and buttons for 'Execute', 'or get result as', 'JSON', and 'or', 'JSON'.

Figure 2.7: The **query box** enables to write a query in order to retrieve or search data.

The screenshot shows the '2. DB-Filters' tab of the CMR interface. It contains four sub-tabs: '2.a User/Type/KW', '2.b Restrictions', '2.c Atoms', and '2d. ASE Traj'. The '2.a User/Type/KW' sub-tab is active, displaying the following fields and buttons: 'User:' with a dropdown menu showing 'martebjo' and an 'Add to query' button; 'Calculation type:' with a dropdown menu and an 'Add to query' button; 'Keywords:' with a dropdown menu showing 'Pt (2521)' and buttons for 'Add to query' and 'Load all'. Below the sub-tabs, there is a 'Query:' section with a hint: '(Hint 1: use * to search for partial keywords; Hint 2: use " for keyword with spaces)'. A large text input field contains the query 'db_user=martebjo'. At the bottom, there is a 'Results per page:' dropdown set to '10', and buttons for 'Execute', 'or get result as', 'JSON', and 'or', 'JSON'.

Figure 2.8: Filters and restrictions that should be applied

which would show all results that the user martebjo uploaded to the database. The restriction tab allows in Fig. 2.9 to restrict the value of a field as for example `ase_potential_energy<0` and the atoms tab enables to add a statement like `atoms=Ag,Cu` which would force the results to contain both atoms Ag and Cu. The last tab labeled ASE Traj provides hints about how to work with ASE trajectory files: trajectories contain multiple steps of a calculation, but in most cases we are only interested in the last one. In this tab you find the needed keyword to select the first or last image.

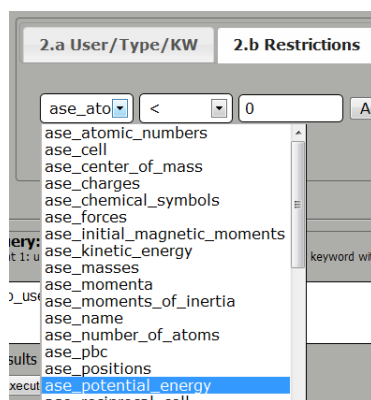


Figure 2.9: When restricting values possible fields are shown, so that one doesn't have to remember or type them.

The enhancement tab as shown in Fig. 2.10 features to add a url field, perform some simple mathematical operation and Fig. 2.11 to add a small Jmol window (showing the atoms) in the table with the results.

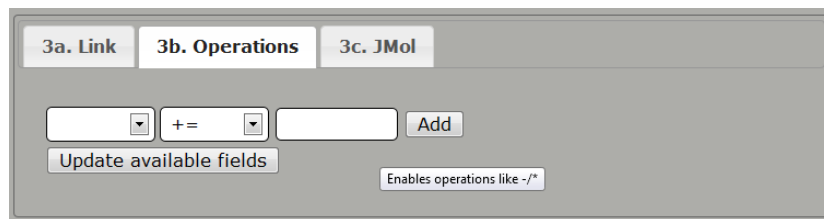
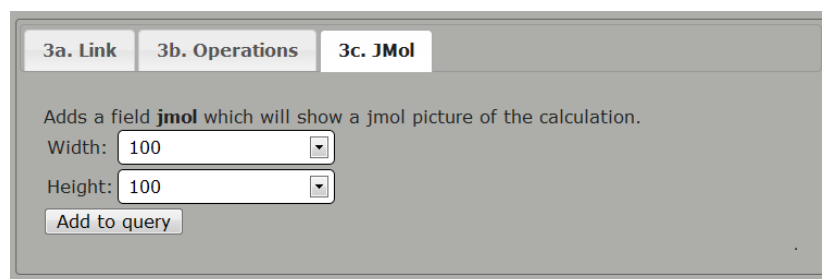


Figure 2.10: The operation sub-tab allows to perform simple mathematical operations on a field.

As a next step (4) it is possible to restrict calculated or enhanced results. This is used for instance when restricting the range of the heat of formation which is calculated and not stored in the database. (It is also possible to restrict fields that are stored in the database but it is far less efficient than using the db-filters.) The other tab defines how to order the results.

In the last step (5) the visualization is defined. If nothing is explicitly chosen



3a. Link 3b. Operations **3c. JMol**

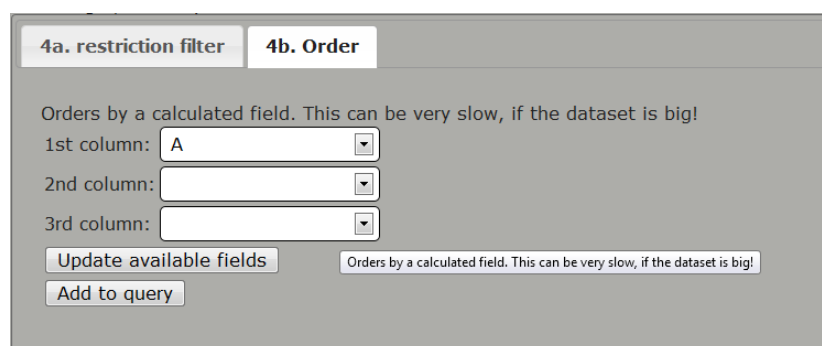
Adds a field **jmol** which will show a jmol picture of the calculation.

Width: 100

Height: 100

Add to query

Figure 2.11: Add Jmol to visualize the atom in the result view.



4a. restriction filter **4b. Order**

Orders by a calculated field. This can be very slow, if the dataset is big!

1st column: A

2nd column:

3rd column:

Update available fields Orders by a calculated field. This can be very slow, if the dataset is big!

Add to query

Figure 2.12: Orders the result according to the selected columns.

the data is shown in a table. The displayed columns can be selected in the table subtab shown in Fig. 2.13.

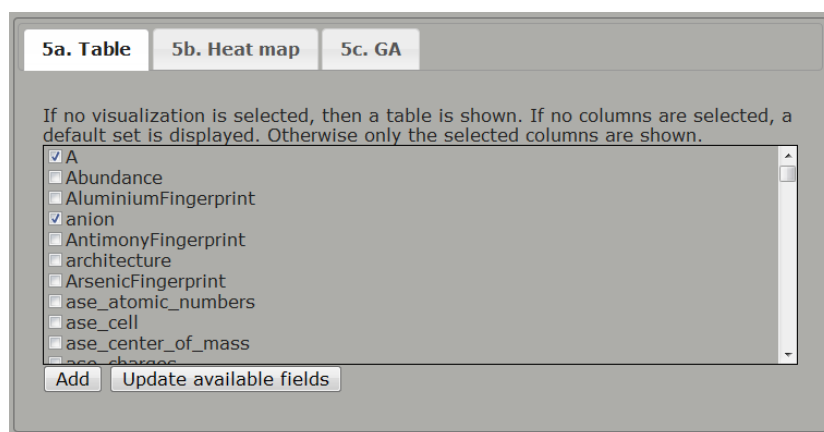


Figure 2.13: By default a table is shown which can be customized by choosing the displayed columns.

The output will look similar to the screenshot in Fig. 2.5.

The Heat map tab allows to create a heat map as shown in Fig. 2.14.

The X/Y axis ticks define the field that is used on the axis and the X/Y sort order can be given a predefined order. Triangle 1 to 4 are the fields that should be shown in the heat-map (although if we choose only one field its a actually shown as a square). The colors for are defined as pairs of value and color names. e.g. 0-yellow defines that the color at 0 is yellow. Values between the defined colors are interpolated while values outside are extrapolated. The two special “values” min and max can be used to denote the minimum and the maximum values respectively. The possible choices for the color names are shown at the bottom of the heatmap definition tab. As a last feature, the an URL column can be defined that is a field that contains a link that the user is transferred to, when clicking into the heat map. Normally this link would show the details of the calculation.

The GA tab, Fig. 2.15 allows to define how the the results of a genetic algorithm run are visualized.

The first choice is to define whether to show the best alleles of a certain generation (best_generation) or the alleles that were calculated in that generation (generation). Then we choose the name of the run. Again it is possible to assign colors for the different values of the fitness which are assigned to the cell in the result table as shown in Fig. 2.15. The columns to display are chosen in the check box list below the other options.

Navigation

By default a table is shown with the result of the created query. The different elements are marked with numbers in Fig. 2.16 and shall be elaborated.

5a. Table **5b. Heat map** 5c. GA

X-axis label: x-axis ordered by the Pe

Y-axis label: y-axis ordered by the Pe

X axis ticks: A

Y axis ticks: B

Triangle 1:(top-right) Value field: e_heat_of_formation Colors*:

Triangle 2:(bottom-left) Value field: Colors*:

Triangle 3: Value field: Colors*:

Triangle 4: Value field: Colors*:

X sort order: Pettifor map

Y sort order:

URL column:

Update available column

Add to query

*Examples for the color

0->white,1->red,7->blue

0->white,0.9->red,2.2->green,4->yellow,8->blue

Figure 2.14: The settings to create a heat map similar to the one shown in figure 2.2

5a. Table 5b. Heat map 5c. GA

Please select which genetic algorithm run to display and the type of the generations to show:.

Display:

Name of run:

Manual color definition*:

Choose the shown allele fields to show:

- ☐ after_hash
- ☐ ase_chemical_symbols
- ☐ ase_positions
- ☐ ase_potential_energy
- ☐ atomic_names
- ☐ atoms
- ☐ before_hash
- ☐ better_than_parents
- ☐ calc_db_file
- ☐ creator
- ☐ db

☐ show parameter name

*Examples for the color choice:
0->white, 1->red, 7->blue

Figure 2.15: Defines how to present a run of the Genetic Algorithm

1 Found 2704 results (0.011036s)

2 < 1 2 3 4 5 6 7 8 9 ... 271 >

3 Restrict keywords:

ABO3 ±(2704)
O3 ±(2704)
max ±(2704)
perovskite ±(2704)
Q ±(2704)
Rb ±(103)
Pt ±(103)
In ±(103)
Ni ±(103)
Re ±(103)
Y ±(103)

4 Jmol

5 id_ref A B anion ase_potential_energy atoms db_keywords downloads

6062 U Re O3 -35.0478 U O Re (1U, 30, 1Re) ABO3 U UReO3 max Q O3 perovskite Re

6070 K Sb O3 -26.0690 Q K Sb (30, 1K, 1Sb) ABO3 K KSbO3 max Q O3 perovskite Sb

6074 Sb Si O3 -29.1894 Q Si Sb (30, 1Si, 1Sb) ABO3 max Q O3 perovskite Sb SbSiO3 Si

6075 Bi Cu O3 -24.0940 Q Cu Bi (30, 1Cu, 1Bi) ABO3 Bi BiCuO3 Cu max Q O3 perovskite

7

8

9

Download information:

ase a python script that retrieves this item from the database and creates an ase atoms object

csv a csv spread sheet with all the data

py a python script that retrieves this item from the database

json all the data in json format

Figure 2.16: Table with the results of the query ABO3 with custom selection of the displayed columns.

- 1 The number of results that were found. The time specifies how long it took to determine the calculations belonging to the result set. This excludes the time to actually build the web page. In some cases results are automatically cached, then the time to read the cached result is displayed.
- 2 The page navigation. Clicking on three will navigate to page three.
- 3 A list of related keywords that are determined by looking at other keywords that results have in the result set. These are counted and added in descending order as suggestions. A click on the keyword will add the keyword to the query and execute it; a click on the “+” beside a keyword will add the keyword to the query, but execute it in a new window.
- 4 The id.ref is the id in the database. This mustn’t be confused with the unique identifier presented in 2.6. A click on it will show all calculation details.
- 5 The atoms contained in this calculation. A click on the atom will add the atom to the query and restrict the results to calculations containing it. If the query restricts to multiple atoms only results containing all the items will be shown.
- 6 The keywords in this calculation. A click on them will add the selected keyword and execute it.
- 7 A link to the query that produced this result. It can be used to get quickly back to the presented view. Note that the query is linked and not the current results: visiting this url will show always an up to date view of the data.
- 8 Download links left to right: A click on the ASE icon will download a script that opens the calculation with the ASE tool “ag” which is capable of displaying the atoms and initiate a new calculation. The CMR link provides a python script template that allows to modify, delete this calculation (an example is shown in the appendix 4.2), and the JSON download provides the data in JSON[20]. Last but not least an xyz[21] file can be obtained. The ASE, and xyz download are however only available for GPAW, Dacapo and ASE created files.
- 9 Jmol view allows to look at the atom in 3D. This third-party tool needs a java plugin for the web browser.

The navigation for the results of the Genetic Algorithm and the heat map are basically the same. For example a click on the id_ref in Fig. 2.15 will open a detailed view of the clicked result.

2.4 Querying and Analysis

Combining the strengths of the PHP/HTML interface and the python interface makes work more efficient: the PHP/HTML interface is good for identifying and verifying data while the python interface provides the ability to filter, print, export and conduct further analysis. It is furthermore capable of connecting to multiple CMR instances, for example a CMR database at CAMD and one at an other institute in order to combine these results in some way. In this section we

0	690681	690831	691041	690801	691056	691003	691066	690662	691114	690909
1	690681	690831	691041	691166	690801	690911	691056	691189	691066	690723
2	690681	690831	691041	691144	691166	690801	690771	691056	691189	691066
3	690681	690831	691021	691040	691041	691141	691166	690801	690638	690925
4	690681	690831	691021	691047	691040	691185	691125	691141	690801	690943
5	690681	690683	690831	691021	691047	691040	690923	691125	690988	690801
6	690681	690701	690949	691044	690961	690916	690683	691040	691125	690672
7	690882	690681	690931	690701	690949	691044	691184	690683	691134	691125
8	690902	691032	690882	690681	690931	690701	690949	690976	690980	690871
9	690742	690902	690659	690989	691032	690882	690681	691190	690735	690980
10	690742	690902	690659	690989	690934	691032	690753	690703	690726	690735
11	690742	690902	690659	690989	690934	691032	690753	690703	690726	690735
12	691060	690902	690659	690989	690934	691032	690753	690703	690726	690735
13	691060	690902	690659	690989	690934	691032	690753	690703	690726	690735
14	690994	691060	690902	690659	690989	690934	691032	690753	690703	690726
15	690844	690994	691060	690902	690659	690989	690934	691032	690753	690703
16	691167	690844	690994	691060	690902	690659	690989	690934	691032	690753
17	690913	691167	690844	690994	691060	690902	690659	690989	690934	691032
18	690913	690913	691167	690844	690994	691060	690902	690659	690989	690934
19	690913	690913	691167	690844	690994	691060	690902	690659	690989	690934
20	690721	690913	690913	691167	690844	690994	691060	690902	690659	690989
21	690870	690721	690913	690913	691167	690844	690994	691060	690902	690659
22	691142	690870	690721	690913	690913	691167	690844	690994	691060	690902
23	691188	691142	690870	690721	690913	690913	691167	690844	690994	691060
24	691188	690876	690919	691142	690748	690981	691135	690721	691169	691179

Figure 2.17: Visualization of the result of a Genetic Algorithm run. The displayed fields (fitness, jmol picture, id) are chosen in the GA tab shown in Fig. 2.15.

1. Data set
2. DB-Filters
3. Enhancement
4. Filters
5. Visualization

1a. Default
1b. Stability and bandgap

By default ALL the data is considered. If this is the aim continue by choosing some filters - otherwise choose a different preselection.

Query:
(Hint 1: use * to search for partial keywords; Hint 2: use " for keyword with spaces)

db_user=dlandis

Results per page: 10

Execute or get result as **html** or json

Figure 2.18: CMR provides a script that executes the same query in the python interface to enable further analysis of the result and json downloads (to the right) the data in the JSON format.

look at how to work efficiently with these tools and show how to get the best performance.

We have already seen how to start a query with the **GroupingAgent**. Another way is to use a **DBReader** object with its **find** method. We have again the possibility to select by name/value pairs, keywords and atoms. There are also choices for excluding keywords/atoms from results as can be seen in Fig. 2.19. All arguments for **find** are optional, but at least one has to be set. The return value is a derived **DataCollection** object which provides functions to create csv files, print to the screen, derive more restricted sets of data, and append **DataCollection** objects to each other.

The most efficient way to query is to make the downloaded results as small as possible with **DBReader.find**. Note that if a restriction to specific columns is omitted, **find** will download all columns of the resulting db-files. Since the download is the bottleneck you are advised to select the columns for larger amounts of data.

Commonly used operations on **DataCollection** objects are

- **DataCollection.select(name, value)** e.g.
`DataCollection.select("db_user", "ivca")` returns a new collection with only items containing data from the user "ivca".
- **DataCollection.select(name, value, comparator)** e.g.
`DataCollection.select("e_heat_of_formation", 0, comparator=Operator("<"))` returns a new collection with only data that has *e_heat_of_formation* < 0.
- **DataCollection.sort(field_name)** e.g.
`DataCollection.sort("db_user")` sorts the collection by the user names.
- **DataCollection.select_if_in_list(name, value)** e.g.
`DataCollection.select_if_in_list("db_keywords", ["ABO3"])` selects items from the list db.keywords that contain the keyword "ABO3".
- **DataCollection.add(other)** returns a new **DataCollection** instance with contains the calling and the data from the other **DataCollection**.

The way that the we used the **DBReader** until now just connected to the default database. In order to connect to a different one we choose a connection name:

```
#connect to default
reader = DBReader()
#connect to third party
other = DBReader("third-party")
```

If the connection is not yet known, a prompt for credentials will show up. To query the db-files that are contained in a directory we can use a **DirectoryReader** object instead of the **DBReader** (see Fig. 2.20). The limit of this approach is that **DirectoryReaders** cannot be used together with agents.

If the connection is not yet known, we are prompted for the necessary credentials.

To make the process of switching from the PHP/HTML to the python interface easier the PHP/HTML interface provides for every query a link to a python interface script that performs the same query if this is possible. (the image link highlighted in figure 2.18). The script contains also hints about how

```
import cmr
from cmr.ui import DBReader

name_op_value_list = [("PotentialEnergy", "<", 0.1)]
name_value_list = [("db_user", "jdoe")]
keyword_list = ["test run"]
not_keyword_list = None
not_atom_list = None
atom_list = ["Cu", "Ar"]
columns = ["PotentialEnergy", "db_user", "db_keywords"]

reader = DBReader()

collection = reader.find(name_value_list=name_value_list,
                        name_op_value_list=name_op_value_list,
                        keyword_list=keyword_list,
                        not_keyword_list=not_keyword_list,
                        atom_list=atom_list,
                        not_atom_list=not_atom_list,
                        columns=columns)

collection.print_table(20)
```

Figure 2.19: Script to retrieve data from database. The query in the PHP/HTML interface would be: `db_user=ivca "test run" atoms=Cu,Ar PotentialEnergy<0.1`.

```
import cmr
from cmr.ui import DirectoryReader

name_op_value_list = [("PotentialEnergy", "<", 0.1)]
name_value_list = [("db_user", "jdoe")]
keyword_list = ["test run"]
not_keyword_list = None
not_atom_list = None
atom_list = ["Cu", "Ar"]
columns = ["PotentialEnergy", "db_user", "db_keywords"]

reader = DirectoryReader("<path>")

collection = reader.find(name_value_list=name_value_list,
                        name_op_value_list=name_op_value_list,
                        keyword_list=keyword_list,
                        not_keyword_list=not_keyword_list,
                        atom_list=atom_list,
                        not_atom_list=not_atom_list,
                        columns=columns)

collection.print_table(20)
```

Figure 2.20: Script to read data/filter db-files from a directory. The query in the PHP/HTML interface would be: `db_user=ivca "test run" atoms=Cu,Ar PotentialEnergy<0.1`

to perform common tasks such as adding keywords, fields, or removing results from the database. The idea is to use it as a starting point to retrieve the desired data for further analysis.

There are many more examples on the CMR wiki at <https://wiki.fysik.dtu.dk/cmr> that show how to work with db-files and DataCollections. The wiki's source can also be retrieved from the CMR source.

2.5 The Command Line Interface

In some situations there is need to just quickly create, modify or submit a db-file. The command line interface (CLI) enables working with db-files in a shell. Additionally, the more powerful commands allow to schedule and submit results to the db-file repository similar to a file versioning tool - however there is no way to go back to older versions with CMR.

Basic Commands

Show all available commands:

```
$> cmr -h
```

To convert any supported file to the db-format `--convert` can be used:

```
$ cmr --convert initial.traj
or
$ cmr --convert *.traj
```

The first example line in the example produces the file `initial.db`, the second converts all `*.traj` in the current directory. Now the produced db-files can be enhanced with keywords, fields and attached scripts and files with the *modify* command, which will show the available sub commands:

```
$ cmr --modify initial.db
cmr version 0.3.2.523M
```

What would you like to do?

```
keywords:  ak:  add keyword(s)
            akr: add keyword(s) recursively: also add to group members
            rk:  remove keyword(s)
            rkr: remove keyword(s) recursively: also remove keywords
                from group members
fields:    av:  add field/variable
            avr: add field/variable recursively: also add to group
                members
            rk:  remove field/variable
            rkr: remove field/variable recursively: also remove
                keywords from group members
scripts:   as:  add script(s)
            rs:  remove script(s)
files:     af:  add file(s)
            rf:  remove file(s)
show:      b:   browse (db-files from of "initial.db")
            s:   print status (only scheduled/added files from
```

```

        "initial.db")
    d:  dump: dump the content of the selected items
    da: dump-all: dump the content of the selected items
        including all group members
commit  c:  commit: write to repository
exit    e:  exit

```

The usage of the commands is intuitive with the exception of *browse* and *print status* that need explaining: *browse* shows all selected db-files while *print status* considers only the status of the files scheduled to be uploaded. In our case we haven't scheduled anything and choose *browse* to see the a few selected columns of what is contained in *initial.db*:

Your choice: b

```

... | db_calculator      | db_user | db_keywords | db_file_name
... | group              | cmr     | NULL        | initial.db
... | ase_trajectory_item | cmr     | ['first ']  | initial.db
... | ase_trajectory_item | cmr     | ['last ']   | initial.db
3 columns.

```

It can be seen that there are three “files”: the items with the *db_calculator* equals “*ase_trajectory_item*” are the steps stored in the original trajectory file while the “*group*” is simply the glue that allows to reconstruct the original trajectory file. There is more information about groups in the section 2.7. In the next step we'd like to add the keywords “NEB” and “initial” to every item:

Your choice: akr

```

Enter one or more keywords (comma separated): NEB,initial
Adding keywords done.

```

The success of the operation can again be checked with **b**. Choose **e** to exit. Back on the command line the db-file can be submitted to the db-file repository with **--commit**

```
$ cmr --commit initial.db
```

Now that the file is submitted it can be found using the python interface or the PHP/HTML interface with the keywords NEB and initial.

Scheduling and Submitting

In some scenarios it is necessary to create a dataset offline and when ready upload it to the database. In case of modifications, the changed part of the dataset can be resubmitted. The commands needed to work this way are **--add**, **--modify**, **--status** **--commit**. CMR then creates a subdirectory with the name *.cmr* that stores the scheduled files: don't modify them manually!

The command **--add** schedules all traj-files in the current directory as to be added to the database. During the selection the files are automatically converted to db-files.

```

$ cmr --add *.traj
cmr version 0.3.2.523

```

```
A  final.traj
A  initial.traj
```

The status can be checked with `--status`:

```
$ cmr --status .
cmr version 0.3.2.523M
```

status	db_orig_fn	db_last_modified	db_keywords	db_hash
A	initial.traj	2011-12-09 23:1...	NULL	78169b4...
A	initial.traj	2011-12-09 23:1...	['first ']	8bf505e...
A	initial.traj	2011-12-09 23:1...	['last ']	eeb029a...
A	final.traj	2011-12-09 23:1...	NULL	548209...
A	final.traj	2011-12-09 23:1...	['first ']	daf1eb2...
A	final.traj	2011-12-09 23:1...	['last ']	a2c288e...

6 columns.

The “A” in front signals that the files are scheduled to be uploaded. It is now possible to edit the files with `--modify` exactly the same way as previously shown:

```
$ cmr --modify initial.traj
```

with the exception that `--status` is used to see the files. When done `--commit` is used in order to upload them to the db-file repository:

```
$ cmr --commit *.traj
```

As soon as the files are available in the database, using `--status` shows the status “DB”

status	db_orig_fn	db_last_modified	db_keywords	db_hash
DB	initial.traj	2011-12-09 23:1...	NULL	78169b4...
M DB	initial.traj	2011-12-09 23:1...	['first ']	8bf505e...
DB	initial.traj	2011-12-09 23:1...	['last ']	eeb029a...
DB	final.traj	2011-12-09 23:1...	NULL	548209...
DB	final.traj	2011-12-09 23:1...	['first ']	daf1eb2...
DB	final.traj	2011-12-09 23:1...	['last ']	a2c288e...

6 columns.

unless the file was modified meanwhile in which case it would be “M DB”.

The CLI has clear advantages when working directly with db-files: the access is fast and easy, but the drawback is that it is not very flexible: all actions are based on file names and are not programmable like in the python interface.

2.6 Modifying Data in the Database

There are a few important details about duplicates, overwriting and removal of data from the database. In a file system there can be only one files with a certain name per directory. CMR works in a similar way, but instead of a file name, a **unique identifier (UI)** is calculated using a hash function over the static part of the included data. The static part is mainly the output of the calculator and

the dynamic part are the user added keywords, fields and attached files. This means that the UI only changes when something unexpected is modified - and prevents therefore overwriting of good with false data. There is a big advantage of having the UI calculated from the data and stored in the db-files and assigned by the MySQL database: the identifier is globally unique and as a consequence data exchange with other databases is possible without conflicts. Additionally the queries written for one database can be used with an other as well.

The right way to update a data in the database is to retrieve it, modify it and then upload it again. The assumption that converting the original file again and uploading the new db-file to the database will overwrite the existing one is wrong. The data is added, but does not overwrite anything because the created UI is different for every conversion.

In seldom cases we have to upload an “old” file and overwrite a newer one. Overwriting in the database created by CMR is similar to a file system where files are overwritten when a file with the same name is added to a directory. Normally there is an option that prevents newer files to be overwritten by older ones. This is the default in CMR: only files with a newer last modified time stamp than the one already in the database can overwrite. As a consequence when uploading older files the last modified time stamp must be updated and this happens, when the db-file is changed or if a the file is *touched*. The easiest way is to use **touch** which is available in the command line interface:

```
cmr —touch *.db
```

and also in the python interface:

```
cmr.touch('name.db')
# or if the file is in memory or just loaded from the database
data = cmr.read('name.db')
data.touch()
data.write('name.db')
```

After touching the db-files need to be re-uploaded.

Using the results of a colleague is easy at CAMD because there is a huge database deployed that everyone can access. Regularly people don't like to modify someone else's data - and it is also risky to not have full control over the data. In this case it is easier to *own* it. Owning means to copy the data and replace the user name with the own one. The command **own** does this and is available in the CLI:

```
cmr —own *.db
cmr —commit *.db # upload to database
```

and also in the python interface:

```
cmr.own('name.db')
# or if the file is in memory or just loaded from the database
data = cmr.read('name.db')
data.own()
data.write('.db') # write directly to database
```

To remove files from the database the command `delete` can be used with a string list of UIs:

```
cmr.delete([string list of UIs])
```

The unique identifiers can be found in different ways depending on the location of the file. The only thing one has to know is that the unique id is called **db.hash** in db-files:

1. From a db-file with the command line interface:

```
$ cmr --dump O2.db
cmr version 0.3.2.524

*****
Calculator/Instance: group/
Hash: 2fff9aa2b8faaaf190ab93356cdb4d2b0acba6c3 <—
Filename: 1.db
Number of group members: 4
*****
static
*****
db_calculator: group
db_file_version: 0.3
db_hash: 2fff9aa2b8faaaf190ab93356cd ... <—
db_last_modified: 2011-12-12 12:45:47.072049
...
```

2. From random db-files in a directory:

```
$cmr -b 0 . --columns db_hash,db_keywords,db_file_name
db_hash          | db_keywords          | db_file_name
486f179507a31b7 ... | NULL                 | ./group.db
2fff9aa2b8faaaf ... | ['N2', 'EMT']        | ./group.db
9c05fea0182582e ... | ['N', 'EMT']         | ./group.db
57e376057e28c7f ... | ['O2', 'EMT']        | ./group.db
2fff9aa2b8faaaf ... | ['N2', 'EMT']        | ./1.db
12882d67a7e6bf4 ... | ['N2', 'EMT', 'first'] | ./1.db
```

Note that the number 0 in the previous command tells to align the columns automatically and the arguments following `--columns` define the column names that should be shown.

3. From a db-file repository:

```
$cmr -b 0 $CMR_REPOSITORY --columns db_hash,db_keywords,db_file_name
...
```

4. From the PHP/HTML interface: a click on *id_ref* will show all data in that calculation including the unique identifier called **db.hash**. See Fig. 2.16 item four to see where to click.

2.7 Groups

It was already shown that the conversion of an ASE trajectory file results in a group storing the contained steps and that the GroupingAgent makes groups for us. It is however often necessary to create groups manually. There are two options: The first one is to include all group members into a single db-file (Fig. 2.21) and the second one is to connect them with the UI (Fig. 2.22).

For data exchange the first approach is suitable: Only a single file needs to be sent, and it contains all data. The second approach is in general recommended especially if the data is already stored in the database.

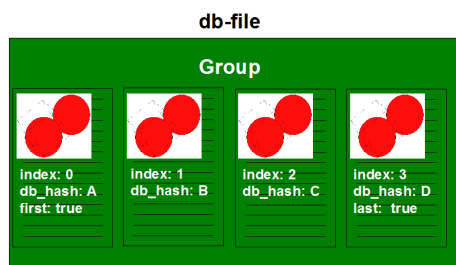


Figure 2.21: A group that contains its members in a single db-file.

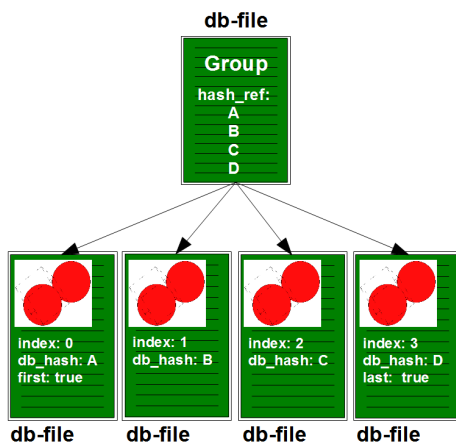


Figure 2.22: A group that references its members. The group identifies its members by the UI which is always called **db_hash** (denoted as A,B,C,D) in the db-files.

Storing the complete members into db-files or referencing them is very similar. The first script stores them:

```
import cmr
member1 = cmr.read('1.db')
member2 = cmr.read('2.db')
```

```

member3 = cmr.read('3.db')

group = cmr.create_group()
group.add(member1)
group.add(member2)
group.add(member3)
group.write('group.db')

    and the second one references them:

import cmr
member1 = cmr.read('1.db')
member2 = cmr.read('2.db')
member3 = cmr.read('3.db')

group = cmr.create_group()
group.add(member1.get_hash())
group.add(member2.get_hash())
group.add(member3.get_hash())
group.write('group_ref.db')

```

2.8 What is actually stored in a db-file?

In this section the content of a db-file shall be discussed. Note however that this section covers only the information needed by a user to work with the db-files and more details can be found in 3.3.3.2.

As shown earlier the data contained in a db-file can be viewed with the command-line interface command `-d`:

```
cmr -d N2.db
```

The content of the db-files depends on what the type of the original file was. Nevertheless there are a few fields that are always contained. Their names start with “db_” in order to prevent conflicts with possible custom fields.

field name	sample value
db_calculator	ase_trajectory_item
db_date	2011-12-12 12:15:54.741326
db_last_modified	2011-12-12 12:15:54.745658
db_hash	356932ea2d6bac7d6433bab8526...
db_location	CAMD Physics, DTU
db_user	dlandis
db_keywords	['EMT','N']
db_description	Calculated N2 molecule with ASE.
db_script	['N2.py']
db_files	['N2.log']

Figure 2.23: Fields that are always present in every db-file.

The meaning of the fields in Fig. 2.23 are as follows: the `db_calculator` field defines the original calculator, other examples besides `ase_trajectory_item` are

gpaw, vasp, gaussian or dacapo. `db.date` stores the creation date of the db-file while `db.last_modified` the one of the last applied modification. The unique identifier is store in the field `db.hash` and the user name in `db.user`. `db.location` should be filled with the department and is defined when installing CMR. The more interesting field are the description (`db.description`), a list of keywords (`db.keywords`), the names of the attached scripts (`db.scripts`) and of other files (`db.files`).

For programs like gpaw and dacapo and as well ASE trajectory files the information that ASE provides is extracted and added as fields. The advantage of using them is that it we know that the units are always in eV and Angstrom. The extracted ASE information from the calculation of a N2 molecule is shown in Fig. 2.24. This information belonging to each atom is interpreted by aligning the arrays: Fig. 2.25.

<code>ase.atomic_numbers</code>	[7, 7]
<code>ase.cell</code>	[[1.0, 0.0, 0.0], [0.0, 1.0...
<code>ase.center_of.mass</code>	[0.0, 0.0, 0.0]
<code>ase.charges</code>	[0.0, 0.0]
<code>ase.chemical_symbols</code>	['N', 'N']
<code>ase.forces</code>	[[0.0, 0.0, -0.005629824829...
<code>ase.initial_magnetic_moments</code>	[0.0, 0.0]
<code>ase.kinetic_energy</code>	0.0
<code>ase.masses</code>	[14.0067, 14.0067]
<code>ase.momenta</code>	[[0.0, 0.0, 0.0], [0.0, 0.0...
<code>ase.moments_of_inertia</code>	[6.9777930978192266, 6.9777...
<code>ase.name</code>	N2
<code>ase.number_of_atoms</code>	2
<code>ase.pbc</code>	[False, False, False]
<code>ase.positions</code>	[[0.0, 0.0, 0.4990868562675...
<code>ase.potential_energy</code>	0.262777484094
<code>ase.reciprocal_cell</code>	[[1.0, 0.0, 0.0], [0.0, 1.0...
<code>ase.scaled_positions</code>	[[0.0, 0.0, 0.4990868562675...
<code>ase.tags</code>	[0, 0]
<code>ase.temperature</code>	0.0
<code>ase.total_energy</code>	0.262777484094
<code>ase.volume</code>	1.0

Figure 2.24: These fields that are extracted with ASE and added to the db-files, if the program supports ASE.

The easiest way to read this information is actually to open the db-file with ase directly and use `view` e.g.

```
from ase.io import read
from ase.visualize import view
data = read('N2.db')
```

	atom 1	atom 2
ase_chemical_symbols	N	N
ase_positions	[0.0, 0.0, 0.499..]	[0.0, 0.0, -0.499..]
ase_masses	14.0067	14.0067
ase_tags	0	0
...

Figure 2.25: Aligning the ASE arrays shown in Fig. 2.24 allows to identify the properties for each individual atom.

```
view(data)
```

or the ASE command line tool **ag**

```
ag N2.db
```

Most of the discussed content should contain sufficient information for most of the users. For more advanced usages the db-file stores also most of the fields available in the original data files with the original variable names (if possible) and units. In general some of the big matrices are ignored in order to keep the size of the db-file small. For GPAW all variables are stored except the following parameters: Projections, AtomicDensityMatrices, NonLocalPartOfHamiltonian, PseudoElectronDensity, PseudoPotential, PseudoWaveFunctions.

People often wonder what the name of the extracted data fields are in detail. It is out of the scope of this work to document these fields that are mostly used by code developers only, but a list of the currently extracted ones is shown in the Appendix 4.1.

2.9 Advanced Task

2.9.1 Publishing results with CMR

The PHP/HTML interface can be used to share data with the public or collaborators. For certain data sets it is useful to restrict the parameters with the purpose of providing a view with specially tailored parameter choices. Examples of such a customization can be found at <http://cmr.fysik.dtu.dk>. Currently it shows the dataset of the case study described in paper [19]. In order to perform such a customization some knowledge in PHP and Javascript are required.

2.9.2 Using a Mapping

When working with different codes it is common that some fields have the same meaning, but a different names. In this case a mapping can be applied to an individual db-file or to a collection of db-files. (The mapping does currently not support to custom defined fields.) Fig. 2.26 shows a complete example script that retrieves some data and maps the name of TotalEnergy to e_tot and the units from eV to hartree. As shown, a dictionary defines the items that should be mapped. The mapping can then be applied to a DataCollection object.

2.9.3 Supporting a new file format

This section explains how to add support for a new file format and implement it as a CMR plug-in. Please note that this section is meant for developers and does not explain all source code in detail. Nevertheless it should be possible to create a converter by following these instructions.

In order to understand how to create a new file format you have to know that CMR supports plug-ins (Fig. 2.27). Plug-ins allow to add functionality to CMR without modifying the CMR source code. In this tutorial we are going to create a converter and a schema plug-in.

2.9.3.1 Step 1: create a converter

We assume that we would like to create db-files from a file format that stores atoms, positions and a identifier. We call the file format newc. An example is shown here:

```
atoms = [6, 8, 8]
positions = [[0, 0, 0], [0, 0, 1.178658], [0, 0, -1.178658]]
name = CO2
```

```

import os
import cmr
from cmr.ui import DirectoryReader
from cmr.tools.units import Units
from cmr.static import CALCULATOR_DACAPO
from cmr.ui.db_reader import DBReader
from cmr.base.mapping import Mapping

mapping_name="custom"

map = {
    'TotalEnergy:' : {
        "name": "TotalEnergy",      # the internal name
        "unit": Units.EV,           # the internal unit
        "type": "double",           # the internal cmr type
        "python_type": "float",     # the internal python type
        "dest_name": 'e_tot',       # the new name
        "dest_unit": Units.HARTREE, # the new unit
    }
}

mapping = Mapping(calculator_name=CALCULATOR_DACAPO,
                  calculator_instance="",
                  name=mapping_name,
                  map=map)
cmr.definitions.register_mapping_range(mapping)

#reader = DirectoryReader(directory, "no_mapping")
reader = DBReader("default")

columns = ["db_user", "e_tot", "TotalEnergy", "atomic_numbers"]

collection=reader.find(name_value_list=[("db_user", "cmr")],
                       columns=columns)
collection.set_mapping(mapping_name) # apply mapping
collection.print_table(0, columns)

```

Figure 2.26: Example of applying a custom mapping to a DataCollection. The field TotalEnergy is mapped to e_tot, and the unit is changed from eV to hartree.

- **agent**: Agents access the database directly and can perform tasks periodically (3.3.3.1)
- **converter**: a converter (3.3.4.1) can read output files of an electronic structure code and write db-files
- **extra_parameters**: are hooks executed before the db-file is written for example during a conversion; these could be used to add extra information about the institute that performed the calculation or add data about the calculation environment or runtime information.
- **sql_schema**: defines how the data should look in the database (section 3.3.2.3 CMR Database)
- **schema**: defines a cmr schema (3.3.3.3 Schemas)
- **test**: a test checks if a specific functionality of CMR works and either succeeds or fails. Tests are executed after the installation or by developers to assure that modifications didn't break parts of the code. It is good practice to provide a test for every converter.

Figure 2.27: CMR plug-in types.

The main task of a converter is to parse foreign file format and store it in a python dictionary in memory. The dictionary can then easily be written to a db-file. Fig. 2.28 shows a complete converter for the newc file format. There are three methods: `accept`, `convert_mem` and `convert`. `accept` returns true, if the file can be converted with that converter, `convert_mem` returns a db-file as an object while `convert` writes a db-file. Please note that the use of the `eval` function is not secure (see 3.3.3.2) and use `safeeval` instead. Another important remark is that we add the fields `ase_positions` and `ase_atomic_numbers` manually. The reason is that the PHP/HTML user interface (3.3.1.3) is able to show the atomic structure, if these fields are present.

2.9.3.2 Step 2: create a CMR schema

The next step is to create a schema (3.3.3.3). Since this task is tedious CMR offers a tool to create the main part of the schema automatically. The only thing that needs to be provided is an example of the data as a dictionary (Fig. 2.29). The next step is to create the complete schema which could look as shown in Fig. 2.30.

2.9.3.3 Step 3: declare the plug-ins

CMR searches plug-in directories for the file `information.py` where it finds the information about the class names of the available plug-ins. For our example the file is shown in Fig. 2.31

```

import cmr
from cmr import logger
from cmr import Log
from cmr import CMRException
from cmr.base.converter import Converter
from cmr.tools.eval import safeeval
from cmr.io.flags import Flags

class NewC2Db:
    @staticmethod
    def accept(filename):
        """returns true, if supported by this converter"""
        return filename.endswith(".newc")

    @staticmethod
    def convert_mem(cmr_params,
                   calculator_instance="",
                   cmr_child_params=None):
        logger.log("Converting newc-file to db-file ...",
                  Log.MSG_TYPE_INFORMATION)
        filename = cmr_params["input"]

        cmr_params["db.cmr-plugin"] = "newc-plugin version 0.1"

        # create dictionary with data
        data = Converter.get_xml_writer(calculator_instance="",
                                       calculator_instance="",
                                       mode=Flags.WRITE_MODE_CONVERT)

        lines = open(filename).read().split("\n")
        for line in lines:
            name, value = line.split("=")
            name = name.strip()
            value = value.strip()
            try:
                value = safeeval(value.strip()) #don't use eval!
            except:
                pass
            data.set_user_variable(name, value)

        data.set_user_variable("ase-atomic-numbers", data["atoms"])
        data.set_user_variable("ase-positions", data["positions"])

        return data

    @staticmethod
    def convert(cmr_params,
               calculator_instance="",
               cmr_child_params=None):
        data = NewC2Db.convert_mem(cmr_params,
                                   calculator_instance,
                                   cmr_child_params)

        data.write(cmr_params)
        data.close()
        return [data.get_hash()]

```

Figure 2.28: newc2db.py: A converter that reads *.newc files.

```
from cmr.tools.functions import make_schema

data =
{'positions': [[0, 0, 0], [0, 0, 1.178658], [0, 0, -1.178658]],
 'ase_positions': [[0, 0, 0], [0, 0, 1.178658], [0, 0, -1.178658]],
 'name': 'CO2',
 'ase_atomic_numbers': [6, 8, 8],
 'atoms': [6, 8, 8]}
print make_schema(data)
```

Figure 2.29: Creates the main part for a CMR schema automatically.

2.9.3.4 Step 4: set environment variables

Add the path of the newc converter to the python path. Also adjust the environment variable CMR_SCHEMA_PATH to point to this location because we're going to create a XML schema in the next step; then add the plug-path to the cmr configuration file that is normally located in `$HOME/.cmr/cmr-settings-file` e.g.

```
plugin_paths=/home/.../newc
```

2.9.3.5 Step 5: create an XML schema

The xml schema allows to validate the db-file. If you made everything right, then you can load the CMR schema as follows and create the XML schema.

```
import cmr
from cmr.definitions import get_calculator_information

calc_info = get_calculator_information("newc" "")
calc_info.create_xsd()
```

Now your CMR installation supports newc files.

```

import os
from cmr import Units
from cmr.base.schema import Schema
from cmr.base.schema import XMLCALCULATOR

class NewCSchema(Schema):

    def __init__(self, name, hash_schema="default"):
        Schema.__init__(self, name, hash_schema=hash_schema)

        calculator = {
            "ase_atomic_numbers":{
                "cmr_name":"ase_atomic_numbers",
                "type":"long_array",
                "optional":"True",
                "python_type":"list",
                "most_inner_python_type":"int",
                "hash":True},
            "ase_positions":{
                "cmr_name":"ase_positions",
                "type":"long_x_array",
                "optional":"True",
                "python_type":"list",
                "most_inner_python_type":"int",
                "hash":True},
            "atoms":{
                "cmr_name":"atoms",
                "type":"long_array",
                "optional":"True",
                "python_type":"list",
                "most_inner_python_type":"int",
                "hash":True},
            "name":{
                "cmr_name":"name",
                "type":"string",
                "optional":"True",
                "python_type":"str",
                "hash":True},
            "positions":{
                "cmr_name":"positions",
                "type":"long_x_array",
                "optional":"True",
                "python_type":"list",
                "most_inner_python_type":"int",
                "hash":True},
        }

        self.append(XMLCALCULATOR, calculator)
        self.apply_default_values()
        self.apply_hash_schema()

    def get_my_file_name(self):
        return os.path.basename(__file__)

```

Figure 2.30: newc_03_20110523.py: Creates the main part for a CMR schema automatically.

```
from cmr import runtime

def info():
    info = [{"name": "newc2db", #module name
            "author": "cmr",
            "type": "converter",
            "classname": "NewC2Db",
            "version": "newcalc-0.3-20110523",
            "module": "newc.newc2db"},
            {"name": "newc_03_20110523", #module name
            "author": "cmr",
            "type": "cmr-schema",
            "calc_name": "newc",
            "calc_instance": "",
            "classname": "NewCSchema",
            "version": "newc-0.3-20110523",
            "module": "newc.newc"}, ]
    return info
```

Figure 2.31: information.py: declares module, class name and version of plug-ins.

Chapter 3

Computational Materials Repository

3.1 Overview

The previous chapter introduced the usage of the Computational Materials Repository (CMR) from the users point of view. This chapter provides more background information about CMR with the focus on how things work and why certain design decisions were taken.

In order to investigate structures, atomization and binding energies, electron transport properties and dynamic processes, one needs to design a model that simulates the behavior at the atomic level. This requires on the one hand a fairly accurate theory to describe the interactions between the atoms and the electrons, and on the other hand, tools to place atoms and optimize their positions according to their interaction. In terms of electronic structure theory Schrödinger described in 1926 already a model that allows to determine the interaction between the atoms accurately. Unfortunately it is far too complex to be solved even on nowadays computers, because it considers all possible electron-electron interactions. Even with simplifications that treat the nuclei and the electrons separately only very small systems can be calculated. An alternative is the density functional theory (DFT) is based on the Hohenberg-Kohn theorems that explain that the many body wave function of the Schrödinger equation can be expressed with the electron density instead. Many of nowadays electronic structure simulators as for example Dacapo[22], GPAW[23] and VASP[24] are based on DFT. The most prominent result that DFT simulators provide is the total energy of a system, and the motion of the atomic positions. Depending on what type of problem is to be solve there are different ways to find solutions with electronic structure simulators. When searching for stable ternary metal borohydrides structures as performed in paper [25] for hydrogen storage, the compounds were first tested in different structures to see which ones are potentially stable.

This is done by relaxing the atoms (relaxing means that the positions of the atoms are optimized so that sum of the forces acting on each atom is minimal). In order to save computation time, all atoms but the hydrogen ones were fixed during the relaxation. The best stable compounds were then completely relaxed in order to see whether the postulated structure was the correct one or if there was a different one. Finally the compounds that allow to bind hydrogen, but not too strongly were selected as candidates and the weight percent of hydrogen was calculated. An other example is to answer the question whether or not an adsorbate would "dock" on a surface or not. In this case the surface and a few layers are modeled and the adsorbate placed on different locations on the surface. Upon comparing the DFT energies, the most favored position can be identified and it can be determined whether the adsorbate would stay. There are many other properties apart from the energy and the updated positions that can be calculated with modern electronic structure codes. An extensive introduction to electronic structure calculations and the theory behind is for example provided by [26].

This chapter starts by explaining the challenges that were faced when designing CMR. In the next step an overview of the system that we built is presented, followed by the description of each component. Then possible improvements are addressed in the outlook and the project is later compared with other implementations.

Without giving away too much details it may be helpful to know that the core of CMR is implemented in Python, is able to use a MySQL database and can provides access to the data with a web server that is programmed in PHP. The database and web server are optional for a minimal system 4.3.1. CMR is usually used on a Linux system.

3.2 CMR Challenges

At the beginning there was the idea to collect data from different electronic structure simulators, and analyze them with VMDF (paper [5]) or an other tool. This was at the very beginning of my PhD. Very soon after, the CAMD Summer School 2008 took place where the researchers participated in a computational materials design project. The generated data had to be collected and analyzed. We were a small team that designed the workflow and I was able to gain invaluable experience by implementing the system to collect and analyze the data. The project and the implementation is outlined in paper [25].

When designing the Computational Repository I had to find compromises between ideas, design wishes, features, resources, performance and prospects. In this section I would like to discuss the faced challenges when building CMR. The first challenge was to figure out what the expected features would be. Because CMR should handle many different kinds of problems I decided to create it

generic with a focus on electronic structure simulators. This has the advantage that it can solve many different problems, but might not be optimal for some of them.

The requirements for CMR were to aggregate, store, monitor, analyze, present and share data from electronic structure simulators. Every of these requirement imposes its own challenges: aggregation requires that CMR can read the output from different codes (3.3.4.1), storing must assure that the collected data can be read also years later even if the original program is unavailable (3.3.3.2), monitoring should provide an almost instant view of the collected data, and the analysis options must satisfy power users that prefer to work with programmable interfaces (3.3.1.2) and casual users that favor more convenient access methods, like a web interface (3.3.1.3). Sharing can be performed through the interfaces (3.3.1) or by sending db-files.

In order to be able to analyze data it was decided that the user should be allowed to enhance the data with custom fields, keywords, and attach scripts and relevant files to provide more background information. Another challenge related to the analysis is the wish to be able to show what data was used to derived results (e.g. which calculations were involved when calculating the chemisorption energy). (2.7).

In terms of usability CMR must not depend on not too many third-party products and provide most of it's functionality out of the box. For example CMR must run independently of a database and a web-server which is especially important when CMR is executed on a computer cluster or on a user's computer with limited access to resources or no database access.

More important requirements are to be able to assure data consistency and prevent data loss, even if processes crash, errors occur or invalid data is tried to be uploaded (3.3.4.2). Another important issue was to find ways to provide the data with an acceptable delay.

The biggest challenge though was to provide a system that is complete and provides the expected functionality - or at least can be extended to provide it. This might seem trivial, but the way from a concept to a working implementation requires to handle a lot of technicalities, special cases and technical limitations.

3.3 System Components and Processes

The data flow of an CMR installation for an institute was already briefly presented in chapter 2. The focus in this section lies on providing more information about the components, show how they interact and explain their purpose. The data flow diagram in Fig. 3.1 highlights the three main type of components: the user interfaces (blue) that are used to interact with the data, the repositories (purple), that store the data, and the db-files (red font) that are containers for transferring and storing data. The arrows denote interactions/processes between the components.

The **conversion** (3.3.4.1)) to **db-files** (3.3.3.2) is a basic functionality of the

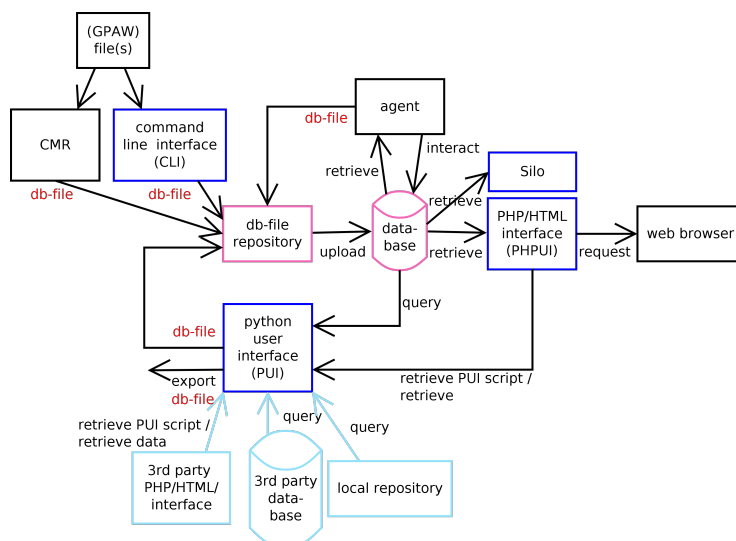


Figure 3.1: Data flow diagram of CMR. Blue: user interface components; purple are repository components; db-files are denoted with red font; black: diverse. Light blue color signifies third-party components.

CMR Python package that is accessible for other components as well. Db-files generally store the data of a single calculation and are used within CMR as a container to transfer or store data. Since db-files are identified by their content and not by their file names and directory locations they are collected in a single directory, the **db-file repository** (3.3.2.1). The data in the db-file repository has normally two purposes: it is kept for long-term storage as a backup and it should be accessible for further analysis. Since it is not very efficient to query a db-file repository, the files are uploaded to the (CMR) database (3.3.2.3). To allow queries to be executed efficiently, the data is rearranged in a better searchable way during the upload. As a consequence of this scheme the data is stored twice in the system: in the db-file repository as long-term storage and in the database for fast access. The figure shows a **local repository** that is not connected to a database as well. This is to show that CMR can also perform basic queries on the db-files as if they were located in a database - although they are a bit slower.

The user interfaces hide the complexity behind intuitive commands and functions. There are four user interfaces with different purposes: the **command line interface (CLI)** (3.3.1.1) is executed in a command shell, is the most basic interface and allows to create and administrate db-files locally as already described in section 3.3.1.1. The **python user interface (PUI)** (3.3.1.2) is the most powerful and provides scriptable access to the local/third-party CMR databases, is able to retrieve data from the local/third-party PHP/HTML interfaces (3.3.1.3) as well as providing basic scriptable access to a local db-file repository. The PUI can

also use agent templates 3.3.3.1 like the GroupingAgent (2.2), but cannot write to the database directly due to access restrictions. The most convenient access to the data is to visit the **PHP/HTML interface** (PHPUI) (2.3,3.3.1.3) with a **web browser**. The user creates a query by selecting criteria. The PHPUI can be programmed to perform more complex analysis [19],3.3.1.3. Because the PHPUI is not scriptable itself it provides a PUI script that executes the same query (see Fig. 2.18) if possible. That script includes code that can be used for further analysis or to modify or download the complete data. The last interface is the HTML/JavaScript **silo user interface** (3.3.1.4) and is mentioned for completeness because it currently works only with the previous version of the CMR database. It is a workbench to analyze data in a web browser that allows to create queries visually. It is capable of working offline with the data which is not possible with the PHPUI. **Agents** (3.3.3.1) are processes that run in the background and perform tasks that normally take a little longer than regular queries. For example some analysis could be performed on all data contained in the database. Currently the agents are used to prepare data needed by the PHPUI.

3.3.1 User Interfaces

The user interfaces enable the user to work with data in a convenient way and hide the complexity of communicating with a repository behind well defined interfaces. CMR features four user interfaces that access the data in different ways.

3.3.1.1 Command line interface (CLI)

Purpose: This interface enables to work with db-files (3.3.3.2) in a command shell. It provides functionality to batch process operations like converting files to db-files, updating content, submitting data to the db-file repository and makes it thereby unnecessary to write programming code for these tasks.

Usage: The command `cmr` enables to batch process files in a command shell. There are two ways to work with this tool. The first one is to use it directly with db-files to create, modify, touch, validate, dump or submit them to the db-file repository. The second mode is to use it to work with a dataset of original output files (e.g. *.traj) that are going to be added to the CMR database. The first step is to select/schedule the files during which they are converted (transparently) to db-files. In the next steps they can be attributed with keywords/fields/script individually or as a group. When the files are ready they can be submitted. If at any later point more keywords are added they can simply be resubmitted.

Usage example(s): Scheduling all *.traj to be uploaded to the database, add keyword “adsorption” and send it to the db-file repository:

```
$> cmr -a *.traj
$> cmr -m *.traj
```

```
...
ak (add keywords)
Your choice: ak

Enter one or more keywords (comma separated): adsorption
$> cmr -c *.traj
```

More examples are presented in section 2.5.

Implementation: The CLI uses the CMR Python package to provide the presented functionality.

Suggested enhancements: This interface could be extended to perform queries similar to the PHPUI (3.3.1.3) and thereby provide quicker access to results.

3.3.1.2 Python User Interface (PUI)

Purpose: The PUI provides scriptable access to the databases and has access to almost all CMR functionality. This includes but is not limited to query data from the local and third-party CMR databases, download data from external CMR web servers running the PHP/HTML interface (3.3.1.3), export db-files and query data in a local repository (3.3.2.2). Last but not least it is able to use agent functionality as for example the GroupingAgent (2.2). The only thing it cannot do is write directly to the database due to access/security restrictions.

Usage: The Python user interface is available, if the CMR Python package is installed. The access to the CMR databases is provided with a special reader (DBReader). This reader allows to search and filter data by atoms, keywords and all other available fields. It is for example possible to retrieve a list of calculations with an energy smaller 0, with the keyword “adsorbate” and with the atom “C”. This list can be further processed: one could just print them, add/remove keywords/fields/scripts, export the data to db-files, create a group or delete them from the database. As mentioned earlier the PUI is not allowed to perform modification on the database directly, therefore when modifying data, it is again sent to the db-file repository. The upload process will detect that the data in the database needs to be updated and perform the modifications. Removing data from the database is similar. CMR’s delete function will create a db-file containing the command to remove that specific data item from the db-file repository and the database. The PUI and the DBReader can also be used to retrieve data from the web interface (PHP/HTML interface), however the unique id (UI: 2.6) must be known. The downloaded data can then be processed (add/remove keywords/fields/scripts) or exported as a db-file to a directory of choice.

The access to the third-party databases is only available if the third-party exposes them to the public, through a VPN or a ssh tunnel. Regularly this is not the case, but when it is available then the procedure and the options are exactly the same as with the local CMR database.

Additionally to the other access methods, the PUI can query any local repository that contains db-files. A special reader named DirectoryReader provides the exact same interface as the DBReader, and therefore the same functionality. The

difference is that the DirectoryReader is slower because it has to load all db-files into memory before being able to execute queries.

The last access option for the PUI is to combine an agent and a DBReader. Agents can be used by regular users, as long as they don't write to the database.

Usage examples: How to use a DBReader: Fig. 2.19

How to use a DirectoryReader: Fig. 2.20

Agent: How to use a GroupingAgent Fig. 2.6

PHP/HTML interface download: Fig. 2.16, item 8 will retrieve a script that downloads the selected item from the PHP/HTML interface using the DBReader.

3.3.1.3 PHP/HTML User Interface (PHPUI)

Purpose: The purpose of the PHPUI is to provide quick access to the data in the database, similar to a state of the art search engine. A query is intuitively written by adding keywords and restrictions on fields. The PHP/HTML interface allows to download data in various formats (JSON, xyz) and provides as well a script that performs the current query in the PUI (3.3.1.2).

Usage: The PHPUI is used to find and view data. Special views are available to visualize runs of the genetic algorithm (Fig. 2.17) or a heat map for the solar light capture[19], Fig. 1.3). The strength of this interface is that queries can be bookmarked, columns of table view chosen according to need and that the query can be composed manually or with assistance and prefilled fields as shown in section 2.3.

Usage examples: The PHPUI is accessed with a web browser. Find a keyword in the list of keywords in the query box (Fig. 2.7) and click execute. The result is displayed as in Fig. 2.16. Now the interface suggests related keywords in the box marked with (3) in the figure which can be used to restrict the results further. If necessary more filters can be added with the help of the tabs in the query box.. When the sought data (set) is found it can download as (8) xyz or JSON file, or by clicking on the ASE-link a script is provided that opens the calculation with the ASE[27] tool **ag**. When the browser is configured to execute the Python script directly then a click on the ASE icon opens **ag**¹ directly. The CMR download link provides a script that retrieves the result as a db-file or allows to modify the content of that db-file. Alternatively, if you'd like to download the whole result set or process it with the PUI, the highlighted link in Fig. 2.18 provides a PUI script enables to do so. (An example of such a script is shown in appendix 4.2.

Implementation: The PHP/HTML interface runs on a web server and translates the requests from the web browser into database language and returns them as HTML back to the web browser. To avoid delays when loading the web page, the related keywords and pre-filled field names in the query box are updated using AJAX[28] (Asynchronous JavaScript and XML) a conglomerate of techniques to send asynchronous requests to the web server and update the web page in the browser dynamically upon arrival of the answer.

¹**ag** is an ASE tool to visualize atomic data(-files), can calculate inter-atomic distances and create new calculations.

Technology: Special care was taken that the PHPUI uses only technology that is available on all commonly used web browsers. It is for example possible to access the web page with an Android² phone or a tablet computer as well. The drawback of the later two devices is however that the user interface requires a mouse for the navigation in the heat map. Security: There are three threats that need to be prevented: PHP code injection, SQL injection and cross-site scripting (XSS). The last one endangers the user only, the first two could have serious consequences for the users and the web server. PHP code injection means that an attacker is able to execute arbitrary commands on the web server. One way to achieve this is by exploiting careless evaluation of user input. Potentially dangerous functions that should not be used with unparsed and escaped input are `eval`, `preg_replace`, `include` or `require`. CMR does not use these functions. Instead it uses `doubleval` to parse double values and `strval` for strings. Preventing SQL injection is a bit trickier because the queries that comes from the PHPUI contain field names and values that are needed in the MySQL query. CMR uses two measures: input validation and escaping of all user input that goes into the query with `mysql_real_escape_string` (escaping means putting extra ‘\’ characters in front of all character that could possibly be understood as a control character to prevent uncontrolled execution of statements.) The last measure taken is to only allow read-only access to the MySQL server from the PHPUI. XSS means to inject a script for example in a form and send it to the server. If the server does not perform any validation and returns the string unparsed it is possible for it to break out of its context and execute script code on the users web browser and thereby for example faking a password field. To prevent XSS all input that is sent back to the web browser must be encoded. PHP offers for this purpose the function `htmlspecialchars`, which is used by CMR. (Reviewed in [29, 30].)

3.3.1.4 Silo Framework, and the SiGUI plug-in

The silo framework is initiated, developed and maintained by Jens Strabo Hummelshøj, at the time of writing a post-doc at SUNCAT Center for Interface Science and Catalysis at Stanford. The current implementation state of the framework and the SiGUI plug-in allows only to communicate with the previous version of the CMR database and is still in an experimental state. This framework is discussed because it is closely related to CMR, and because the author of this thesis has created SiGUI, a plug-in for visual query selection.

Purpose: The Silo web interface runs in a web browser and provides the user with tools to analyze data on his own computer, even when disconnected from the Internet. Its design allows to create, use and share plug-ins with third-parties easily. A screenshot is shown in Fig. 3.2.

Usage: With silo it is possible to administer and browse through calculations, work offline, views structures and data in tables in a web browser.

Silo vs. PHPUI: Both interfaces are accessed with a web browser. The difference

²Android is a Linux-base operating system that runs on phones and tablet computers.

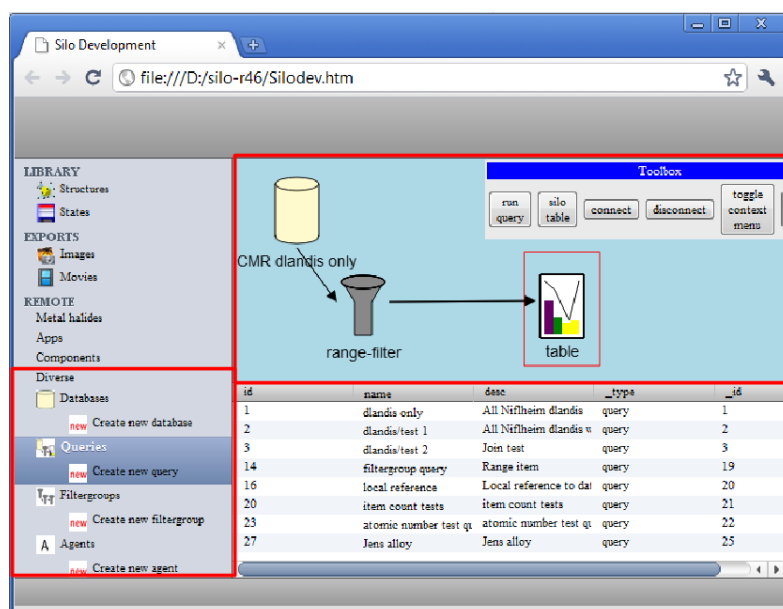


Figure 3.2: Screenshot of silo and SiGUI. The red parts are created by the SiGUI plug-in (see below).

is where the user interface runs and in what language they are implemented: silo runs in the web browser and is mainly programmed in JavaScript while the PHPUI runs on a web server and is programmed in PHP. In terms of functionality silo is rather a desktop application while PHPUI is more a search machine.

3.3.1.4.1 SiGUI, a silo plug-in

Purpose: SiGUI (**S**imple **G**raphical **U**ser **I**nterface) is a plug-in for Silo that was created by the author of this thesis. Its designation is to work like VMDF[5] and allow to create queries visually in the silo environment.

Figure: see Fig. 3.3

Usage: In order to use SiGUI silo needs to be opened in a web browser as shown in Fig. 3.2. SiGUI provides a few basic filters as for example “select” where $x \text{ op } y$ where x is a variable name, op an operation like $>$, $<$, $=$ and y a value. New higher level filters are created by coupling a few basic filters as shown in figure 3.3 that shows that a range filter is composed of a $a > \dots$ and a $a < \dots$ filter.

Limitations: Currently SiGUI works only with the previous of the CMR database, and created filters cannot be exchanged with third parties. The implementation level is a prototype.

SiGUI vs. VMDF: The main difference between SiGUI and VMDF is that SiGUI is implemented in JavaScript and that the filters can be composed of other filters. The main idea is to have a few general filters that can be used as a base for the

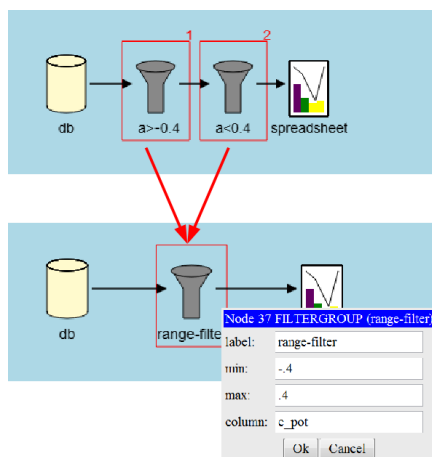


Figure 3.3: SiGUI allows to create queries in a graphical user interface. Series of the filters like an $a > \dots$ and an $a < \dots$ can be merged into a single filter.

creation of more complex filters which then can be used by even higher level filters. An example is shown in figure 3.3. The depicted range filter is composed of a $a > \dots$ and $a < \dots$ filter. The missing values *min* and *max* will be asked in the property window of the range-filter.

3.3.2 Repositories

The repositories are used to store data and make them accessible. There are three different types of repositories: the **db-file repository**, the **(CMR) database** and the **local repositories**. The db-file repository is the location, where users deposit the db-files that they would like to access from the CMR database later. A specially dedicated upload process (3.3.4.2) takes care of validating and copying the data to the CMR database. The local repositories are just directories that contain db-files. CMR allows to perform basic queries on the contained db-files as if they were located in a CMR database. In this section we will provide some more background information about these repositories, and how and where they are used.

3.3.2.1 db-file repository

Purpose: The db-file repository acts as a data store for data that is to be analyzed or should be stored long-term.

Usage: The db-file repository is regular directory that hosts db-files and everyone (on a file system) can write his/her db-files that he/she would like to add to the CMR database. The location is identified by the environment variable `CMR_REPOSITORY`, but regularly db-files are directly written to the db-file repository by specifying the name ".db" during conversion. The conversion process is

discussed in section 3.3.4.1.

Technical Details: There are a few practical issues that needed to be handled for our file system. The first one is that people will submit their results with arbitrary file names and this results in file name conflicts; the second one is that storing a huge number of files results in a noticeable delay when listing the content of the directory to check for newly added db-files (100000 files took approx. 3 minutes on our NFS files server). For this reason the uploading process (3.3.4.2) renames them first and then moves them into the “inbox” which writes the file into a dedicated subdirectory if the current destination has more than 1000 files. For example the file `100059_01434...db` should be copied to the directory `inbox/`. If `inbox/` contains already 1000 files then these files are distributed into subdirectories by their first index. Our file would go to `inbox/1`. When the directory `inbox/1` has more than 1000 files it is split again, but this time the second character defines the directory. In our case the file would end up in the directory `inbox/1/0`. This pattern is followed until a directory is found with less than 1000 files. This technique is applied for all directories that store db-files within the db-file repository and assures quick access times.

3.3.2.2 local repository

A local repository is a directory that contains db-files.

Purpose: Store db-files locally and perform basic queries on the db-files in that directory.

Usage: When retrieving data from third-party databases or when creating new datasets the db-files can be stored in a local repository in order to be updated/-modified before being uploaded to the database. An other use is to filter db-files without a database. `DirectoryReader` enables basic to execute basic queries and works the same way as `DBReader`. The performance is however considerably slower, because the `DirectoryReader` has to read all db-files into memory before being able to filter the data.

Usage example: An example of how queries are performed on a local repository with a `DirectoryReader` is shown in Fig. 2.20.

3.3.2.3 CMR Database

In this work the *CMR database* is often referred as *database*. The technical correct term would be a MySQL database with the CMR database schema.

Purpose: The CMR database allows data to be uploaded from the db-file repository and enables fast queries and downloads of data including attached scripts and files.

Usage: The upload process (3.3.4.2) loads data from the db-file repository to the CMR database. The only other component that has write access to the database is the agent (3.3.3.1). Disallowing write access to casual users ensures that they cannot break anything and that all data is uploaded consecutively.

Implementation: This section provides more information about the internal organization of the data in the database. CMR allows to use customized database

schemas, but we concentrate on the default schema that is used by the PHPUI.

The first challenge is to find a **database schema** that allows storing of heterogeneous data in a relational database without knowing exactly what kind of analysis should be performed. (If we knew the analysis requirements, then we could derive an optimal table layout with standard database approaches as for example with the entity relationship model[31].)

We will first show why the the straight forward approach fails and then the CMR solution. We use a relational MySQL database (see section 3.4.3) that stores data in tables consisting of named columns and rows. The straight forward approach of storing all data in a single indexed table will not work because (A) the row-size is limited to 65535 bytes and the column-count to 4096 columns or less depending on the data that is stored in it[32] (B) adding a new column to the table means to add it to every row that is already in the database and is expensive (C) users can add arbitrary fields of arbitrary types with the same name, which will eventually result in type conflicts for the same column name. Fig. 3.4 is used to illustrate the problems.

		m columns			
n rows	id	Ekin	Epot	valid	...
	1	.1	.01	0	...
	2	.2	.02	False	...
	n

Figure 3.4: A table with n rows and m columns. The upload of the second piece of data results in a conflict because there can be only one variable type per column.

(A) The row-size limit is quickly reached especially if strings are stored: if 128 bytes per string were reserved, then there would be space for 512 columns which is quickly reached considering that data from multiple simulators and an arbitrary number of custom fields can be added. (B) The set of columns cannot be determined beforehand because users can create new custom field names at any time. Every new column will result in a table reorganization that modifies all n already existing rows. This can result easily result in a delay of several minutes depending on the size of the table. (C) The above table shows a type conflict: an earlier version defined **valid** to be an integer value while the following uses boolean. In MySQL every column has an fixed type that cannot be altered. Therefore the upload with the boolean value would fail.

Since we don't know how the data will be analyzed and we cannot create a huge sparse table because the fields cannot be identified beforehand, a pragmatic approach was chosen; the variables are divided by type and written into one single table. An example is shown in Fig. 3.5. This approach results in a 5 tables, one for strings, doubles, dates, booleans, and one for arrays.

This schema allows fast querying, but when retrieving a whole db-file with j fields it would result in j database join operations which are expensive and

double					
id	name	value			
1	Ekin	.1			
1	Epot	.01			
1	valid	0			
2	Ekin	.2			
2	Epot	.02			

boolean		
id	name	value
2	valid	False

Figure 3.5: Example of how variables are stored in the CMR database. The data that belong together have the same id, denoted by the colors blue and green. The data is the same as in Fig. 3.4, but organized in a different way.

slow. Therefore the db-file is uploaded and converted to a text string in the JSON[20] format which is more efficient than *j* joins in terms of database CPU usage. The JSON string is for example used by the PHPUI when showing a whole calculation or when downloading a whole db-file with the PUI.

Technical Details: The challenge that the processes face when querying a database schema that is organized as shown in Fig. 3.5 are that the type of the variable has to be known in order to execute the query on the right table. When the user writes the query `valid=0` how do we know in which table to look? One option is to determine the type from the input variable and then conclude that the type that is sought is the same. In this case 0 is an integer therefore the conclusion is that we look in the numeric value, but this approach fails, if we look for `surface=001`. This is because 001 is interpreted as 1 which is an integer, but what is actually meant is the string “001”. The PHPUI considers additionally the total number of entries of “surface” in the double, string, boolean, ... tables. Regularly the result would be that “surface” is located in the string table. Since the statistical guess is weighted more than the user’s input type the choice would be correct. In some rare cases the type guess is wrong. In this case the user must either change the name of the variable, or adjust the type.

Disk Memory usage: The database schema’s memory usage is not optimal. It consumes about five to six times the amount of memory than the db-files use on disk. The reason is that all data in the db-files is compressed and in the database it is stored uncompressed. The MySQL database supports compression of columns, but unfortunately the process is not transparent. (This means the syntax to access a compressed column is different from accessing an uncompressed column.) Therefore a migration to compressed columns is difficult and implies some downtime for all CMR database installations. This issue should probably be addressed during a bigger restructuring.

3.3.3 Other Components

3.3.3.1 Agents

Agents are processes that run periodically on the server and work directly on database and perform data analysis tasks or prepare information for special

views or customized tables. Since users don't have write access to the database they can generally not run agents. However some of the agents don't write to the database and these can be used by the user.

Purpose: Agents are used to execute heavy database operations and cache data for the user interfaces. They can also be used to create customized views or to update a customized database schema.

Usage: The agents are executed periodically on the server. The frequency is defined by the developer of the agent. All registered agents (registered as a plug-in) are executed periodically. By default two agents are enabled: the first one converts the atomic numbers in the database to atomic names. The second agent creates a table that shows how often each keywords occurs and another table with the data items per user. These tables are used by the PHPUI to avoid executing these expensive queries over and over.

Usage Examples: How to use the GroupingAgent is shown in section 2.2

Background: *Software agents* can be classified according to *autonomy* (act without human interaction), *social ability* (interaction with other agents), *responsiveness* (should perceive their environment and respond to changes) and *proactiveness* (take initiative to reach goals) (reviewed in [33]). The CMR agents are currently only performing tasks autonomously and are therefore the simplest kind of agents. The interface however doesn't prevent users from implementing smarter agents.

3.3.3.2 Db-files/cmr-files

The db-files store the data of a single or multiple calculation, user-defined fields and keywords, and attached scripts and files. The content of db-files is well defined: every type of calculator gets its own CMR schema (3.3.3.3) that defines exactly what fields are stored, their type and optionally the unit. Since the data is stored in XML it is possible to validate the content against a XML schema(3.3.3.3). It is possible to create new special CMR schemas for specific purposes. For example one could create a schema for slab calculations that would ensure the user provides a field containing the number of layers and reject the db-file if it doesn't.

At the beginning we named the file(-format) db-files (database files). The choice of the name is not very advantageous though because many other products use the same name. Therefore we are about to name them cmr-files.

Purpose: Db-files were designed for data-exchange with third parties and for long-term storage. The content can be verified against a schema. The content of a db-file is as close as possible to the original file in terms of field names, types and units so that people can recognize the data in the usual format.

Usage Examples: Db-files are either created by converters (3.3.4.1), by agents (3.3.3.1) or by the PUI (3.3.1.2). The content of db-files can be viewed with the CLI (3.3.1.1) or with the PUI.

Background on long-term storage: Storing data long-term means to be able to ensure that the data remains readable. When we look at the data collected at CAMd/DTU, then the data that survived many years was most often stored

in spreadsheets, without information about the original calculation - or in the the CAMd database that was maintained by H. L. Skriver[6]. Most the binary output files of calculations that were created over 4 years ago were deleted or cannot be read any more because the file-format was changed or new methods are used. Therefore we have decided to store the data in in a human readable form.

The criteria that db-files must satisfy are as follows: the content should be human and machine readable, no external libraries should be necessary to read and write them, the content should be small and verifiable. Human readable implies that people can extract data intuitively in contrast to machine readable which means that it is possible to write a preferably simple parser for the given data structure. Not using third-party software allows to be free in the design and maintenance of the file-format, enables to implement changes quickly and avoid thereby being overruled or forced to adapt due to third-party decisions. Cost for disk space is still an issue nowadays, even though the prices are constantly falling[34]: therefore db-files have to be small. In order to assure that the data is complete the db-files must be verifiable which implies that schemas can be created for them that define the content.

We decided to use XML[35, 36] (eXtended Markup Language) to represent the data and stores it in a single tar.bz2[37, 38] compressed container. The compressed container is not only used to make the files smaller, but also because it allows to add more files as for example the script that performed the calculation and other output files. Having the script is great, when wondering how the result was obtained.

XML is human- and machine readable, widely supported and all major programming languages provide source code to parse XML data. XML can be used with XML schemas that define the content of an XML file. For example one can specify which variables are mandatory and of what variable type they have to be (e. g. integer, double, array, string). Validating a XML file against a XML schema checks if the syntax is correct and checks whether the constraints are violates. Fig. 3.6 shows an extract of data stored in the XML format.

The currently best available alternative to the db-file format is ETFS[39]. It was designed for electronic structure and crystallographic data. Since it bases on the NetCDF (Network Common DataForm) library[40, 41] it also capable of storing fields (variables), keywords and attributes like a name, a unit or even a range to give the data additional meaning. Since the implementation is in C and Fortran it is very fast. The current version 4.1.3 is fully backwards compatible according to the official web page([41]) which needs to be accentuate, because it is not a given feature. Although this format is attractive it doesn't fit CMR because it requires third-party software.

Size of db-files: It is hard to provide an average size of a db-file because users can add arbitrary data and attach files. Therefore a few examples shall be provided of three different datasets. They are shown in Fig. 3.3.3.2.

Security: Because db-files are used for data exchange with third parties it needs to be assured that they cannot be abused. An attacker could for example try to inject code. (This means to embed code into a db-file that is executed on the

```

...
<calculator>
  <TotalEnergy><double>-586.442</double></TotalEnergy>
  <AtomicNumbers>
    <long_array length="2">
      <long>6</long>
      <long>8</long>
    </long_array>
  </AtomicNumbers>
...
</calculator>
...

```

Figure 3.6: The content is intuitively obvious as follows: *TotalEnergy* is equal -586.442 and *AtomicNumbers* is an array containing 6 and 8. “double”, “long_array” and “long” are the names of the variable types.

Ternary alkali-transition metal borohydrides with Dacapo:

program	Dacapo
number of db-files	5581
total size	285 MB
average db-file size	51 KB
attached files	no

Perovskite Metal Oxides

program	GPAW
number of db-files	5621
total size	128 MB
average db-file size	20 KB
attached files	no

Automatic data collection of a students data

program	237 Dacapo, 1704 trajectory files, 358 GPAW
number of db-files	2299
total size	977 MB
average db-file size	400 KB
attached files	yes: *.py, *.traj, *.txt

Figure 3.7: Example of collected data in db-files and their average file sizes.

victims machine when the file is opened.) The goal could be to get passwords, modify or delete content, install a virus, et cetera. In Python all functions that interact with the operating system or allow dynamic code execution (e.g. `exec`, `eval`, `os.system`, `os.popen`, `execfile`, `input`, `compile`) should not be used with data from the third-party input files. The only function that CMR uses when parsing the third-party file is `eval`. This function is used to convert strings to numbers or arrays. Although potentially dangerous it is easy to sanitize `eval` by manually set the local global environment as unavailable. This prevents `eval` from executing any code. The small code below demonstrates the effect: first the dangerous version and then the sanitized one.

```
# dangerous
eval("open('/tmp/new', 'w').write('alert')")
# sanitized
eval("open('/tmp/new', 'w').write('alert')", {'__builtins__':None}, {})
```

For more information about `eval` and python in general see [17].

3.3.3.3 Schemas

There are two kinds of schemas: the CMR schema and the XML schema. (The database schema is of different nature and discussed in section 3.3.2.3.) The CMR schema is the main schema and defines for every calculator (and calculator version) the fields, their types and units, if available. The XML schema is derived from the CMR schema and used in order to perform the db-file validation (3.3.4.3).

Purpose: The CMR schema is the description of the data in the db-file and is used during the extraction and during the validation.

Implementation: The CMR schema is implemented in Python and defines all variables stored in the corresponding db-file. Fig. 3.8 shows an extract of the GPAW schema. These CMR schemas can be adapted to specific program versions or they can be designed loose to accept several versions with the same schema. It is also possible to derive a schema from another one and make it specific for a certain type of calculation. One could imagine to make one for a chemisorption calculation where the atoms in the surface have to be put as separate fields.

Example: section 2.9.3

3.3.4 Processes

3.3.4.1 Conversion to /writing of a db-file

The conversion process takes an original output file and converts it to a db-file.

Purpose: Create db-files and define keywords/fields and attach scripts/files.

Usage: The creation of the db-files is performed by the user with the CLI (3.3.1.1) or with the `cmr` Python package.

Process: First the file type is detected and the correct converter determined. The converter is responsible to select the correct version of the CMR schema (3.3.3.3) depending on the version of the input file. When the data is read from the

```

class GPAWSchema(Schema):

    def __init__(self):
        ...

        calculator = {
            ...
            'PoissonStencil': {
                "optional": False,
                "type": "string",
                "python_type": "int_or_str"},
            'XCFunctional': {
                "optional": False,
                "type": "string",
                "python_type": "str"},
            'Epot': {
                "optional": False,
                "unit": Units.HARTREE,
                "type": "double",
                "python_type": "float"},
            'AtomicNumbers': {
                "optional": False,
                "desc": "A list of atomic numbers.",
                "type": "long_array",
                "python_type": "numpy.array",
                "most_inner_python_type": "int"},
            'RubidiumFingerprint': {
                "optional": True,
                "type": "string",
                "python_type": "str"},
            ...

```

Figure 3.8: Extract of the GPAW's cmr-schema. `PoissonStencil`, `XCFunctional`, `AtomicNumbers` and `RubidiumFingerprint` are variables. `type` defines the internal type of the variable, `python_type` the original type, `atomic_numbers`. The field `unit` defines that `Epot` has the unit Hartree and the `desc` is a descriptive string of what of the value that the variable holds.

original output file, all types are mapped to types that CMR supports internally (see Fig. 3.9 for a list). If an exception occurs during the conversion the partial db-file is written to disk and an error message shown to the user. The reason for writing the partial file is that DFT calculations are expensive and data should not just be lost without being inspected.

Technical Details: The converters were designed as plug-ins (2.9.3). The reason

Python type	Int. type	Python	Int. Repr.
int	long_type	1	1
float	decimal_type	1.2	1.2
complex	array	1+2j	1+2j
str	string	"john"	"john"
list	array	[[1,2,3], [4,5,6]]	[[1,2,3], [4,5,6]]
tuple	array	((1,2,3), (4,5,6))	[[1,2,3], [4,5,6]]
numpy.array	array	numpy.array([])	[]

Figure 3.9: In order to enable CMR to run anywhere without third-party software, CMR uses only standard python types internally. During the conversion process or when writing fields to a db-files the types are mapped to the internal ones. The following data types are supported: boolean, int, float, complex, string, datetime.datetime, list and arrays of the previously mentioned data types. All other known data types are mapped to the internal representation.

is to assure that license incompatibilities can be circumvented. CMR uses GPL. If a converter for a new file format should be created that causes a license conflict (because of some dependencies or restrictions), it is possible to distribute the converter under a different license than GPL. The drawback is just that CMR cannot distribute this converter and the users have to install it separately.

Conversion examples: Fig. 2.4, 2.2, Fig. 2.4 and Fig. 2.3

3.3.4.2 Upload

The upload process consists of three independently executed tasks: (1) moves files from the db-file repository (3.3.2.1) to an inbox directory, (2) validates the db-file and (3) uploads the data to the database (3.3.2.3).

Purpose: The upload process copies files from the db-file repository to the database.

Usage: The upload tasks are scheduled to run periodically (every 10 seconds) to check, if new files were written to the db-file repository, if so task (1) renames and moves the files to an internal "inbox" folder. Task (2) checks the validity of the new arrivals and moves invalid files to a dedicated folder and the valid ones to the "valid" folder. Task (3) gets notified and uploads the new data to the database.

Implementation Details: Task (3) is responsible to check, if the newly arrived data is new, a duplicate, an updated or outdated data. This is determined with the unique identifier and the last modified time stamps as explained in section 2.6.

3.3.4.3 Validation

Purpose: The validation of db-files assures their completeness, ensures that all required fields are present and checks the attached files for potentially dangerous file types.

Usage: The db-files are validated before they are accepted in the db-file repository (3.3.4.2). It is also possible to initiate a validation by the user. During the validation the content of the db-file is inspected: first the XML file is checked against the XML schema (3.3.3.3). It is valid, if the syntax is correct and all mandatory fields are present with the expected type. The second steps assures that all attached files are actually contained in the db-file and the last step makes sure there are no files with potentially malicious file extensions (see security).

Usage Examples: CLI: `cmr --validate`, PUI: `import cmr; cmr.validate("...")`

Security: Db-files are normally uploaded to the CMR database and all their content will be available in the PHPUI. This includes the attached scripts and files. An attacker could therefore attach a malicious file to a db-file and hope that someone executes it. To prevent this scenario, the validation rejects db-files that contain files with the following forbidden file extensions: 386, bin, cmd, com, crt, dll, exe, fon, html, js, jse, lnk, msi, msp, oxc, pif, reg, scr, sys, vbe, vbs, vs, wsc, wsf, wsh, xpi.

3.4 Tools

3.4.1 Motivation

CMR relies on a few third-party software products and technologies. In order to be useful for CMR they must be available for free³, popular (in the sense of people like to use that product because of its advantages over competitors) and expected to be maintained and supported for a long time. Especially the expected lifetime is important because CMR is used as long-term storage for the data. Because CMR should be maintained and developed by different people, the software should be well known and commonly used. This section elaborates why Python, Apache/PHP and MySQL were selected. Products that need to be licensed for money are not discussed.

3.4.2 Python

CMR must be able to read output files of different code during data aggregation, provide a scriptable user interface for finding and analyzing data, access collected and stored results long-term and communicate with the database. This can only be achieved with a programming language is popular among scientists, easy to learn, supports object oriented design, doesn't cost money and is available on most platforms and operating systems. Popular languages in science are C/C++,

³Please note the distinction between free and *open-source software*[42]: free means there is no payment involved while open-source allows to study, modify and in some cases also to redistribute modified code.

Java, Python or Fortran[43, 44].

We shall briefly look at the advantages/disadvantages of these languages. The fastest and most memory efficient code is produced by C/C++ and Fortran because the code is directly translated into machine language and the memory allocation/deallocation is performed by the programmer. This implies that the source code is larger and more complex than with Java and Python where a garbage collector takes care of the memory and the compiler creates Bytecode. Bytecode is an intermediate representation of the source code that is computer architecture independent, but has the drawback that it still has to be compiled which results in regularly slower execution times than binary programs. Nevertheless Python is popular among scientists due to its attractive extensions like NumPy (N-dimensional array objects), SciPy (efficient numerical routines) and matplotlib (2D and 3D plots) that simplify programming more than comparable libraries in other languages. (Reviewed in [43, 44, 45, 46, 47])

We chose Python because it fulfills the requirements and is a good compromise between ease of use and efficiency. The mentioned Python extensions are however not used by the implementation of CMR in order to keep it as independent as possible.

3.4.3 MySQL

Scalability is important for CMR. It should be able to support projects of small sizes as well as all data generated by a whole institute. To support the later it uses a database management systems often just called database. The requirements for the database software are to be able to store electronic structure calculations, search efficiently for fields and keywords, provide interfaces for Python and PHP, be easy to install and maintain and available on the many platforms.

The nowadays most commonly used type is the *relational database* which was invented by Edgar F. Codd and published in 1970[48]. The reason of the wide acceptance until now is that the concept is relatively simple: data is organized in tables with rows and columns. One row is referred to as a tuple consisting of attributes (the values in the columns). A collection of tuples with the same attributes (table) is called a relation, hence the name relational database. There are two other database types: the *NoSQL databases* (not only SQL) and the *object oriented database management systems*. The latter is capable of storing complete objects in a database and works therefore hand in hand with object oriented designed software. The NoSQL databases enjoy increased interest because they promise to solve mature shortcomings of the relational databases like the problems of scalability (relational databases are only vertically scalable⁴), throughput, high set up and maintenance cost, finding compromises between

⁴By vertical scalable is meant upscaling: a single machine is upgraded with disk, memory, CPU in contrast to horizontally scalable which would mean adding more machines[49].

reliability and performance are addressed. (Reviewed in [49, 50, 51].)

For CMR we have decided to use MySQL, a relational database. Although the other database types have very attractive features the choice is difficult, depends on the exact usage and it is not clear how long they will be maintained which is a big drawback. The main arguments why MySQL was chosen were availability and popularity.

3.4.4 Apache/PHP

To visualize results in text and graphics CMR needs a web server, a programming language that runs on the web server and can use an interface to access/search a database and create content in *HyperText Markup Language* (HTML) that is presented to the user.

One option for the server side programming language is the general-purpose language Perl which provides a special library for web development and many other add-ons. Add-ons allow to choose the components that are needed, but make the installation on different platforms cumbersome. Java, other general-purpose language enables building a web-servers through its JSP (JavaServer Pages) and servlets, but requires quite some programming effort. Although Java and Perl are great languages they have a steeper learning curve compared to PHP (Hypertext Preprocessor) which is easy to learn, provides many build-in features such as MySQL database access, can be integrated in many web servers and widely applied. (Reviewed in [52, 53, 54]).

According to *Netcraft survey*[55] (in 2011) the Apache server is the most popular web server since 1996 followed by Microsoft's IIS server and nginx. nginx does not support PHP directly which results in lower performance of PHP - although the web server is generally very fast[56]. Reasons for the popularity of Apache are its feature richness: secure communication (SSL), support for server side programming (PHP languages, modules for authentication), availability on many different platforms, and that it is open-source software[57].

The choice of Apache and PHP is obvious due to their positioning in the market. They are both feature rich, exist for long, are open-source and actively developed.

3.4.5 Tools Summary

The third-party components for CMR, Python, Apache/PHP and MySQL were chosen according the requirements, popularity and availability on different platforms. Because all of these software packages are actively developed and maintained we expect them to provide a stable foundation for CMR. Since CMR is modular it would be possible to exchange third-party software and for example use a different database software.

3.5 Outlook

There are many ideas how to make CMR better in terms of increased performance, better user interfaces and improved usability. In this section the ones with the supposedly greatest effect will be shown and elaborated.

3.5.1 Improvements of upload and validation

The upload process (3.3.4.2) and validation (3.3.4.3) can be improved in several ways. During the validation and the uploading of the db-files, a log is kept about the performed actions. This log could be extended to store as well the associated file owner. As a consequence it would be possible to identify the account from which malicious or fake db-files were submitted. Another improvement that goes into the same direction is to create a process that notifies users about changes and problems during the upload process. To improve validation time of the db-file the current process could be enhanced to allow parallel validation. (Technical detail: it needs a separate process, Python 2.4 threads are not good enough.)

3.5.2 Improvements of the PHP/HTML user interface

The web interface lacks ways to directly update the data. Often it is desired to add/remove a keyword or put a description. Currently the user needs to use the provided PUI script that downloads the data, then it is modified and uploaded again. A desired solution would be to enhance the PHP/HTML user interface to permit to change keywords and custom fields directly. Another desired feature is user authentication. This would enable control of who has access to the database. Depending on how it is implemented it would also allow to restrict the accessible data. The last nice improvement would be, to create a custom user interface more easily, so that the user can adapt the view to his needs.

3.5.3 A submission tool

The public database does not allow direct data upload, and this is good due to security reasons. Nevertheless there should be a way that allows people to submit their data when they finished a paper. At CAMd we upload the data to the local database first. When the paper is published the data is retrieved and copied to the public database.

External users have however no access to the internal database. Alternative solutions are: (1) add the upload capability to the PHP/HTML interface, (2) set up an ftp server (this is a special server allows to upload/download files) and give access on request (3) implement a custom tool.

The simplest method is to use a ftp server because it allows to handle files efficiently. The customization of the PHP/HTML interface or a fully customized tools are nice to have but they are not really necessary.

3.5.4 CMR in the Cloud

Cloud computing is a service that provides resources such as memory, computing hours, hosting of databases or virtual machines. The name comes from seeing the components only through the cloud: it is unclear (and unimportant) for the subscriber what hardware is used and how it is organized exactly. Depending on whether an institute has its own computing resources and an administrator, it might be cheaper and more scalable to host the CMR database in the cloud.

3.6 Conclusion

CMR meets the requirements to provide a framework that is able to aggregate, store, monitor, analyze, present and share data from electronic structure calculations. In order to perform data exchange, a new file format (db-files/cmr-files) was created that does not depend on third-party software and enables long-term data storage. CMR provides different kinds of user interfaces that enable users with different expertise to access data: the PHP/HTML user interface provides access within a web browser, the Python user interface allows scriptable access and the command-line interface makes working with db-files easier. The implementation works out of the box after the installation, is built from well-known stable components, is held very general and can therefore handle most of the desired tasks. However, due to the generality it doesn't work optimally in all situations. If the constraints for queries get too complicated, then a custom arrangement of the involved data in the database is required. This arrangement could be created with an agent that automatically updates it when new data is added to the database. (This agent has to be implemented by the user.) The Computational Materials Repository shows how to solve the problems related to collection, presentation and analysis of data from electronic structure codes and proofs that it works by providing a working implementation.

Chapter 4

Appendix

4.1 Extracted Values from Codes

The following lists shows the parameters that are extracted and stored in the db-files. Note that not all output files contain always all variables. The parameters are ordered by capitalization and then alphabetically.

Dacapo

AtomicNumbers, BZKpoints, ChargeMixing, ConvergenceControl, Date, Density_WaveCutoff, DynamicAtomAttributes, DynamicAtomForces, DynamicAtomPositions, DynamicAtomSpecies, DynamicAtomVelocities, EigenValues, ElectronicBands, ElectronicMinimization, EnsembleXCEnergies, EvalFunctionalOfDensity_XC, EvaluateCorrelationEnergy, EvaluateExchangeEnergy, EvaluateTotalEnergy, ExcFunctional, FermiLevel, IBZKpoints, InitialAtomicMagneticMoment, Keywords, KpointWeight, MagneticMoment, NetCDFOutputControl, OccupationNumbers, PlaneWaveCutoff, StructureFactor, SymmetryGeneratorOrder, TotalEnergy, TotalFreeEnergy, TotalStress, TypeNLProjectorl, TypeNLProjectorm, UnitCell, UseSymmetry, User, ase_atomic_numbers, ase_cell, ase_center_of_mass, ase_charges, ase_chemical_symbols, ase_dipole_moment, ase_forces, ase_initial_magnetic_moments, ase_kinetic_energy, ase_magnetic_moment, ase_magnetic_moments, ase_masses, ase_momenta, ase_moments_of_inertia, ase_name, ase_number_of_atoms, ase_pbc, ase_positions, ase_potential_energy, ase_reciprocal_cell, ase_scaled_positions, ase_tags, ase_temperature, ase_total_energy, ase_version, ase_volume, atomdos_angular_channels, atomdos_energygrid_size, atomdos_projections, atomdos_radial_orbs, hardgrid_dim1, hardgrid_dim2, hardgrid_dim3, longstring, max_projectors_per_atom, number_BZ_kpoints, number_IBZ_kpoints, number_ionic_steps, number_of_bands, number_of_dynamic_atoms, number_of_spin, number_of_symm_gen, number_plane_waves, real_complex, softgrid_dim1, softgrid_dim2, softgrid_dim3

Gaussian

Atomic_numbers, Basis_set, Charge, Chemical_formula, Compact_data, Com-

puter_system, Date, HF, Method, Multiplicity, Person, Positions, Sequence_number, Title, Type_of_run

GPAW

ActiniumFingerprint, AluminiumFingerprint, AmericiumFingerprint, AntimonyFingerprint, ArgonFingerprint, ArsenicFingerprint, AstatineFingerprint, AtomicNumbers, BZKPoints, BariumFingerprint, BasisSet, BerkeliumFingerprint, BerylliumFingerprint, BismuthFingerprint, BoronFingerprint, BoundaryConditions, BromineFingerprint, CadmiumFingerprint, CaesiumFingerprint, CalciumFingerprint, CaliforniumFingerprint, CarbonFingerprint, CartesianForces, CartesianPositions, CeriumFingerprint, Charge, ChlorineFingerprint, ChromiumFingerprint, CobaltFingerprint, Converged, CopperFingerprint, CuriumFingerprint, DataType, DensityConvergenceCriterion, DensityError, DysprosiumFingerprint, Ebar, Eext, EigenstateError, EigenstatesConvergenceCriterion, Eigenvalues, EinsteiniumFingerprint, Ekin, EnergyConvergenceCriterion, EnergyError, Epot, ErbiumFingerprint, EuropiumFingerprint, Exc, FermiLevel, FermiWidth, FermiumFingerprint, FixDensity, FixMagneticMoment, FluorineFingerprint, FortranTransport, FranciumFingerprint, GadoliniumFingerprint, GalliumFingerprint, GermaniumFingerprint, GoldFingerprint, GridSpacing, HafniumFingerprint, HeliumFingerprint, HolmiumFingerprint, HydrogenFingerprint, IBZKPointWeights, IBZKPoints, IndiumFingerprint, InterpolationStencil, IodineFingerprint, IridiumFingerprint, IronFingerprint, KohnShamStencil, KryptonFingerprint, LanthanumFingerprint, LawrenciumFingerprint, LeadFingerprint, LithiumFingerprint, LutetiumFingerprint, MagnesiumFingerprint, MagneticMoments, ManganeseFingerprint, MaximumAngularMomentum, MendeleviumFingerprint, MercuryFingerprint, MixBeta, MixClass, MixMetric, MixOld, MixWeight, Mode, MolybdenumFingerprint, NeodymiumFingerprint, NeonFingerprint, NeptuniumFingerprint, NickelFingerprint, NiobiumFingerprint, NitrogenFingerprint, NobeliumFingerprint, NumberOfBandsToConverge, OccupationNumbers, OsmiumFingerprint, OxygenFingerprint, PalladiumFingerprint, PhosphorusFingerprint, PlatinumFingerprint, PlutoniumFingerprint, PoissonStencil, PoloniumFingerprint, PotassiumFingerprint, PotentialEnergy, PraseodymiumFingerprint, PromethiumFingerprint, ProtactiniumFingerprint, RadiumFingerprint, RadonFingerprint, RheniumFingerprint, RhodiumFingerprint, RubidiumFingerprint, RutheniumFingerprint, S, SamariumFingerprint, ScandiumFingerprint, SeleniumFingerprint, SetupTypes, SiliconFingerprint, SilverFingerprint, SodiumFingerprint, SoftGauss, StrontiumFingerprint, SulfurFingerprint, Tags, TantalumFingerprint, TechnetiumFingerprint, TelluriumFingerprint, TerbiumFingerprint, ThalliumFingerprint, ThoriumFingerprint, ThuliumFingerprint, Time, TinFingerprint, TitaniumFingerprint, TungstenFingerprint, UnitCell, UnnilhexiumFingerprint, UnnilpentiumFingerprint, UnnilquadiumFingerprint, UraniumFingerprint, UseSymmetry, VanadiumFingerprint, XCFunctional, XenonFingerprint, YtterbiumFingerprint, YttriumFingerprint, ZincFingerprint, ZirconiumFingerprint, architecture, ase_atomic_numbers, ase_cell, ase_center_of_mass, ase_charges, ase_chemical_symbols, ase_dipole_moment, ase_dir, ase_forces, ase_initial_magnetic_moments, ase_kinetic_energy, ase_magnetic_moment, ase_magnetic_mo-

ments, ase_masses, ase_momenta, ase_moments_of_inertia, ase_name, ase_number_of_atoms, ase_pbc, ase_positions, ase_potential_energy, ase_reciprocal_cell, ase_scaled_positions, ase_tags, ase_temperature, ase_total_energy, ase_version, ase_volume, energyunit, gpaw_dir, history, lengthunit, nadm, natoms, nbands, nbzkpts, nfinegptsx, nfinegptsy, nfinegptsz, ngptsx, ngptsy, ngptsz, nibzkpts, norbitals, nproj, nspins, pid, pid, units, version

VASP

EDIFF, EDIFFG, EMAX, EMIN, ENAUG, ENMAX, IALGO, IBRION, ICHARG, ISIF, ISMEAR, ISPIN, ISTART, LORBIT, MAGMOM, NBANDS, NELECT, NELM, NELMDL, NELMIN, NPAR, NSW, POTIM, PREC, SIGMA, SYSTEM, VOSKOWN, atomic_numbers, atoms, cellf, date, divisions, e_0_energy, e_fr_energy, e_wo_entrp, elementnames, forces, kscheme, location, mass, platform, program, pseudo, stress, subversion, types, user, valence, volf, voli

4.2 PHPUI script to continue analysis in the PUI

This section presents a script downloaded a CMR web server from the PHP/HTML interface. The query in the web-interface was `db.user=ivca atoms=Cu ABO3` and the same query is run when the script is.

```
# This data was downloaded from CMR
# Link to whole dataset: http://cmr.fysik.dtu.dk/...
# *****
# If you use data from others don't forget to cite!
# *****

from cmr.ui import DBReader
from cmr.tools.type_converters import DateConverterBase

def date_convert(string):
    try:
        return DateConverterBase.convert(string)
    except:
        return string

connection_name="default"
reader = DBReader(connection_name, db_prefix="ivca")

# Converting atomic names to atomic numbers
from cmr.static import atomic_numbers
atom_name_list=["Cu"]
atoms = [atomic_numbers[atom_name] for atom_name in atom_name_list]

# add here the columns that should be downloaded and shown:
# please note that you should never use id_ref - use db.hash instead
columns = ["db_calculator"]
collection = reader.find(keyword_list=["ABO3"],
                        atom_list=atoms,
                        name_op_value_list=[('db_user', '=', 'ivca')],
                        columns=columns)

collection.print_table(columns=columns)

#####
# Add keywords and fields to selected items and submit
# it to the database again
#####
# add keywords and other fields:
# result_dict = collection.get_as_dict(columns=["db_hash"])
# hash_list = result_dict["db_hash"]
# for hash in hash_list:
#     data = reader.retrieve(hash)
#     #add a keyword to all
#     data.get("db_keywords", []).append("my_new_keyword")
#     #add a user-defined field verified
#     data.set_user_variable("verified", True)
#     #write it to the database
#     data.write(".db")
```

```
#####  
# CREATE a group with selected item  
#####  
# result_dict = collection.get_as_dict(columns=["db_hash"])  
# hash_list = result_dict["db_hash"]  
# collection.create_group(cmr_params={"db_keywords":["new group"]})  
  
#####  
# DELETE selected data from database  
#####  
# import cmr  
# result_dict = collection.get_as_dict(columns=["db_hash"])  
# hash_list = result_dict["db_hash"]  
# cmr.delete(hash_list)
```

4.3 Deployment Examples and Dependencies

Depending on the available resources and the size of the project, CMR can be installed differently. Three examples are going to be presented.

In order to create db-files, often the original calculator and all its dependencies must be installed as well. Currently this applies to Dacapo, GPAW and ASE trajectory files.

Here is a list with the explanation what the components are for. More information on how to install these components can be found in the CMR wiki <https://wiki.fysik.dtu.dk/cmr> in the *Server Setup* section.

- Python 2.4.3 or newer
The Python interpreter is needed because CMR is implemented in Python
- (python-simplejson extension - included in Python 2.6 and newer)
When db-files are uploaded to the database also a representation in json is created that is retrieved upon querying and needs to be parsed - which is slower if Python-simplejson is not installed
- CMR
CMR can be installed from the repository or from the RPMS created by Marcin Dulak. More information can be found in the CMR wiki.
- CMR database upload scripts, available from the CMR wiki
Scripts that check the db-file repository for new files, perform the validation and the upload to the database
- MySQL 5.0.77
The MySQL database - for user systems the MySQL client is enough, for server the MySQL server needs to be installed.
- MySQL-python or PyMySQLPyMySQL[58]
Python cannot natively connect to the MySQL database and needs one of the above extensions to do so.
- Apache server (httpd)
The webserver.
- PHP
Needed to create the web-pages.
- PHP json extension
Needed to parse the json representation.
- cron
If the data should be automatically uploaded to the database a *cron job* executes the upload script in a certain interval.
- xmllint
Xmllint is used to perform the XML validation. CMR can also validate db-files without it, but this is not automatically supported, therefore it is required when uploading data to the database

4.3.1 Minimal

A minimal environment can handle up to a couple of thousand calculations. The limiting factor is the time it takes to read the db-files or the db-file cache

from the repository to memory and may vary from machine to machine. The simplejson extension is optional but strongly recommended to install .

- Python 2.4.3 or newer
- (python-simplejson - included in Python 2.6 and newer)
- CMR
- (xmllint - if files should be XML-validated)

4.3.2 Medium

A medium size environment can handle many calculations as long as the query is performed by the database; the bottleneck is the download of the db-files from the MySQL database. Because of the redundancy in the MySQL database of the uploaded data there is considerable amount of extra disk space needed - approximately a factor five of the disk size of the db-files.

- Python 2.4.3 or newer
- (python-simplejson - included in Python 2.6 and newer)
- CMR
- CMR database upload scripts, available from the CMR wiki
- MySQL 5.0.77
- MySQL-python or PyMySQLPyMySQL[58]
- (cron - if data should be automatically uploaded to the MySQL database)
- xmllint

4.3.3 Institute or high quality database

The deployment for an institute or for the high quality database is the same.

- Python 2.4.3 or newer
- (python-simplejson extension - included in Python 2.6 and newer)
- CMR
- CMR database upload scripts, available from the CMR wiki
- MySQL 5.0.77
- MySQL-python or PyMySQL
- Apache server (httpd)
- PHP
- PHP json extension
- cron

4.4 Inside a db-file

A minimal db-file is a compressed tar.bz2 container containing a file named "info.xml". "info.xml" contains the data of either a calculation or the definition for a group of calculations. Figure 4.1 shows an example of XML code to demonstrate how the data is stored in "info.xml":

```
...
<calculator>
  <TotalEnergy><double>-586.442477162</double>
</TotalEnergy>
  <AtomicNumbers>
    <long_array length="2">
      <long>6</long>
      <long>8</long>
    </long_array>
  </AtomicNumbers>
...
</calculator>
...
```

Figure 4.1: XML extract from a GPAW calculation as it is stored within "info.xml" in a db-file. The content is as follows: *TotalEnergy* = -586.442477162 and *AtomicNumbers* = [6, 8]. "double", "long_array" and "long" are the names of the internal type.

The XML-file is divided into seven paragraphs as illustrated in 4.4:

- **static:** data that is present in every db-file like the date and the user name
- **extras:** information that describe the calculation like keywords and a description
- **calculator:** all data from the original calculator output that was selected to go into the db-file
- **user:** custom data that the user can chosen freely
- **ASE:** data from retrieved from the ASE interface; only available, if ASE is used with this file format; this will contain some duplicate data that is already present in the calculator paragraph - but the units will be eV and the coordinate absolute and not relative.
- **history:** when db-files are modified and the hash value changes the previous hash value should be kept as a backup; in the future there might be ability to search also for older hash values
- **runtime:** information about how the calculation was run. Currently none of the converter collects this information automatically

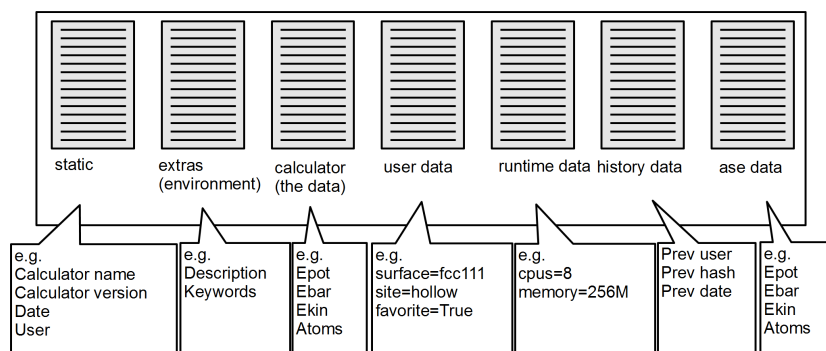


Figure 4.2: Detailed content of a db-file. The shown variable names are of descriptive nature to show what goes where. The ASE data seems to be duplicate - but it actually presents the data in a way that the users are used to see it in terms of units and absolute coordinates.

Bibliography

- [1] National Center for Biotechnical Information (NCBI). <http://www.ncbi.nlm.nih.gov/>, 2012-01-31.
- [2] T Hubbard, D Barker, E Birney, G Cameron, Y Chen, L Clark, T Cox, J Cuff, V Curwen, T Down, R Durbin, E Eyraas, J Gilbert, M Hammond, L Huminiecki, A Kasprzyk, H Lehtvaslaiho, P Lijnzaad, C Melsopp, E Mongin, R Pettett, M Pocock, S Potter, A Rust, E Schmidt, S Searle, G Slater, J Smith, W Spooner, A Stabenau, J Stalker, E Stupka, A Ureta-Vidal, I Vastrik, and M Clamp. The Ensembl genome database project. *Nucleic Acids Res.*, 30(1):38–41, JAN 1 2002.
- [3] Protein Data Bank (PDB). <http://www.pdb.org/>, 2012-01-31.
- [4] Human Genome Project. http://www.ornl.gov/sci/techresources/Human_Genome/home.shtml, , 2012-01-31.
- [5] T R Munter, D D Landis, F Abild-Pedersen, G Jones, S Wang, and T Bli-gaard. Virtual materials design using databases of calculated materials properties. *Comput. Sci. Discovery*, 2(1):015006, 2009.
- [6] CAMd Database. <http://www.camd.dtu.dk/Forskning/Databases.aspx>, 2012-01-31.
- [7] AflowLib: Ab-initio Electronic Structure Library. <http://www.aflowlib.org/>, 2012-01-31.
- [8] Alec Belsky, Mariette Hellenbrandt, Vicky Lynn Karen, and Peter Luksch. New developments in the Inorganic Crystal Structure Database (ICSD): accessibility in support of materials research and design. *Acta Crystallogr., Sect. B*, 58(3 Part 1):364–369, Jun 2002.
- [9] Inorganic Crystal Structure Database (ICSD). <http://www.fiz-karlsruhe.de>, 2012-01-31.
- [10] Gary Yuan and François Gygi. ESTEST: a framework for the validation and verification of electronic structure codes. *Comput. Sci. Discovery*, 3(1):015004, 2010.

- [11] Anubhav Jain, Geoffroy Hautier, Charles J. Moore, Shyue Ping Ong, Christopher C. Fischer, Tim Mueller, Kristin A. Persson, and Gerbrand Ceder. A high-throughput infrastructure for density functional theory calculations. *Comp. Mater. Sci.*, 50(8):2295 – 2310, 2011.
- [12] MaterialsGenome. <http://www.materialsgenome.org/>, 2012-01-31.
- [13] Quixote. <http://quixote.wikispot.org/>, 2012-01-31.
- [14] Sam Adams, Pablo de Castro, Pablo Echenique, Jorge Estrada, Marcus Hanwell, Peter Murray-Rust, Paul Sherwood, Jens Thomas, and Joe Townsend. The Quixote project: Collaborative and Open Quantum Chemistry data management in the Internet age. *Journal of Cheminformatics*, 3(1):38, 2011.
- [15] Quantum Materials Informatics Project. <http://www.qmip.org/>.
- [16] S. R. Bahn and K. W. Jacobsen. An object-oriented scripting interface to a legacy electronic structure code. *Comput. Sci. Eng.*, 4(3):56–66, may 2002.
- [17] Python. <http://python.org/>, 2012-01-31.
- [18] Comma-separated values. http://en.wikipedia.org/wiki/Comma-separated_values, 2012-01-31.
- [19] Ivano E. Castelli, Thomas Olsen, Soumendu Datta, David D. Landis, Soren Dahl, Kristian S. Thygesen, and Karsten W. Jacobsen. Computational screening of perovskite metal oxides for optimal solar light capture. *Energy Environ. Sci.*, 5:5814–5819, 2012.
- [20] F.P. Miller, A.F. Vandome, and J. McBrewster. *JSON: Computer, Human-readable Medium, Data Structure, Associative Array, Douglas Crockford, Internet Media Type, Serialization, Ajax (programming), XML, JavaScript, Ecma International*. VDM Publishing House Ltd., 2009.
- [21] XYZ file format. <http://openbabel.sourceforge.net/wiki/XYZ>, 2012-01-31.
- [22] B. Hammer, L. B. Hansen, and J. K. Nørskov. Improved adsorption energetics within density-functional theory using revised Perdew-Burke-Ernzerhof functionals. *Phys. Rev. B*, 59(11):7413–7421, Mar 1999.
- [23] J. J. Mortensen, L. B. Hansen, and K. W. Jacobsen. Real-space grid implementation of the projector augmented wave method. *Phys. Rev. B*, 71:035109, Jan 2005.
- [24] G. Kresse and J. Furthmüller. Efficiency of ab-initio total energy calculations for metals and semiconductors using a plane-wave basis set. *Comp. Mater. Sci.*, 6(1):15 – 50, 1996.

- [25] J. S. Hummelshøj, D. D. Landis, J. Voss, T. Jiang, A. Tekin, N. Bork, M. Duřak, J. J. Mortensen, L. Adamska, J. Andersin, J. D. Baran, G. D. Barmparis, F. Bell, A. L. Bezanilla, J. Bjork, M. E. Björketun, F. Bleken, F. Buchter, M. Bürkle, P. D. Burton, B. B. Buus, A. Calborean, F. Calle-Vallejo, S. Casolo, B. D. Chandler, D. H. Chi, I Czekaj, S. Datta, A. Datye, A. DeLaRiva, V Despoja, S. Dobrin, M. Engelund, L. Ferrighi, P. Frondelius, Q. Fu, A. Fuentes, J. Fürst, A. García-Fuente, J. Gavnholt, R. Goeke, S. Gudmundsdottir, K. D. Hammond, H. A. Hansen, D. Hibbitts, Jr. E. Hobi, J. G. Howalt, S. L. Hraby, A. Huth, L. Isaeva, J. Jelic, I. J. T. Jensen, K. A. Kacprzak, A. Kelkkanen, D. Kelsey, D. S. Kesanakurthi, J. Kleis, P. J. Klüpfel, I Konstantinov, R. Korytar, P. Koskinen, C. Krishna, E. Kunkes, A. H. Larsen, J. M. G. Lastra, H. Lin, O. Lopez-Acevedo, M. Mantega, J. I. Martínez, I. N. Mesa, D. J. Mowbray, J. S. G. Mýrdal, Y. Natanzon, A. Nistor, T. Olsen, H. Park, L. S. Pedroza, V Petzold, C. Plaisance, J. A. Rasmussen, H. Ren, M. Rizzi, A. S. Ronco, C. Rostgaard, S. Saadi, L. A. Salguero, E. J. G. Santos, A. L. Schoenhalz, J. Shen, M. Smedemand, O. J. Stausholm-Møller, M. Stibius, M. Strange, H. B. Su, B. Temel, A. Toftelund, V Tripkovic, M. Vanin, V Viswanathan, A. Vojvodic, S. Wang, J. Wellendorff, K. S. Thygesen, J. Rossmeisl, T. Bligaard, K. W. Jacobsen, J. K. Nørskov, and T. Vegge. Density functional theory based screening of ternary alkali-transition metal borohydrides: A computational material design project. *J. Chem. Phys.*, 131(1):014101, 2009.
- [26] R.M. Martin. *Electronic structure: basic theory and practical methods*. Cambridge University Press, 2004.
- [27] Atomic Simulation Environment (ASE). <https://wiki.fysik.dtu.dk/ase/>, 2012-01-31.
- [28] A.T. Holdener. *Ajax: the definitive guide*. Definitive Guide Series. O'Reilly, 2008.
- [29] C. Snyder, T. Myer, and M. Southwell. *Pro PHP Security: From Application Security Principles to the Implementation of XSS Defenses*. Apress Series. Apress, 2010.
- [30] M. Nystrom. *SQL Injection Defenses*. O'Reilly shortcuts. O'Reilly, 2007.
- [31] Peter Pin shan Chen. The Entity-Relationship Model: Toward a Unified View of Data. *ACM T. Database Syst.*, 1:9–36, 1976.
- [32] *MySQL 5.0 Reference Manual*. 01 2012.
- [33] Nick Jennings and Michael Wooldridge. Software Agents. In *IEE Review*, pages 17–20, 1996.
- [34] Cost of Hard Drive Storage Space. <http://ns1758.ca/winch/winchest.html>, 2012-01-31.

- [35] XML Technology. <http://www.w3.org/standards/xml/>, 2012-01-31.
- [36] Extensible Markup Language (XML) 1.1 (Second Edition). <http://www.w3.org/TR/2006/REC-xml11-20060816/>, 2012-01-31.
- [37] Tar. <http://www.gnu.org/software/tar/>, 2012-01-31.
- [38] bzip2. <http://bzip.org/>, 2012-01-31.
- [39] Xavier Gonze, C.-O. Almbladh, A. Cucca, D. Caliste, C. Freysoldt, M. A. L. Marques, V. Olevano, Yann Pouillon, and M. J. Verstraete. Specification of an extensible and portable file format for electronic structure and crystallographic data. *CoRR*, abs/0805.0192, 2008.
- [40] R. Rew and G. Davis. NetCDF: an interface for scientific data access. *IEEE Comput. Graphics Appl.*, 10(4):76–82, jul 1990.
- [41] NetCDF. <http://www.unidata.ucar.edu/software/netcdf/>, 2012-01-31.
- [42] M.R. Overly, Pike, and Inc Fischer. *The open source handbook*. Pike & Fischer, 2003.
- [43] A. Tveito, H.P. Langtangen, B.F. Nielsen, and X. Cai. *Elements of Scientific Computing*. Texts in Computational Science and Engineering. Springer, 2010.
- [44] Mathieu Fourment and Michael Gillings. A comparison of common programming languages used in bioinformatics. *BMC Bioinf.*, 9(1):82, 2008.
- [45] H. Schildt. *Java The Complete Reference, 8th Edition*. The Complete Reference. McGraw-Hill Companies, Inc., 2011.
- [46] M. Gregoire, N.A. Solter, and S.J. Kleper. *Professional C++*. John Wiley & Sons, 2011.
- [47] Fernando Perez, Brian E. Granger, and John D. Hunter. Python: An Ecosystem for Scientific Computing. *Comput. Sci. Eng.*, 13:13–21, 2011.
- [48] E. F. Codd. A relational model of data for large shared data banks. *Commun. ACM*, 13:377–387, June 1970.
- [49] S. Tiwari. *Professional NoSQL*. John Wiley & Sons, 2011.
- [50] S.W. Dietrich and S.D. Urban. *Fundamentals of Object Databases: Object-Oriented and Object-Relational Design*. Synthesis Lectures on Data Management. Morgan & Claypool, 2011.
- [51] Rick Cattell. Scalable SQL and NoSQL data stores. *SIGMOD Rec.*, 39:12–27, May 2011.
- [52] M. Doyle. *Beginning PHP 5.3*. John Wiley & Sons, 2011.

-
- [53] J. Gerner, E. Naramore, M. Owens, and M. Warden. *Professional LAMP: Linux, Apache, MySQL and PHP5 Web Development*. John Wiley & Sons, 2006.
 - [54] T. Butzon. *PHP by example*. By Example. Que Pub., 2002.
 - [55] Web Server Survey. <http://news.netcraft.com/survey/>, 2012-01-31.
 - [56] T. Tomlinson and J. VanDyk. *Pro Drupal 7 Development, Third Edition*. Apress Series. Apress, 2010.
 - [57] The Apache Software Foundation. *Apache HTTP Server 2.2 Official Documentation - Volume III. Modules (A-H)*. Fultus Corporation, 2010.
 - [58] PyMySQL. <https://github.com/petehunt/PyMySQL/>, 2012-01-31.