The database

Tools

The database (DB) was setup in SQLite (version 3). SQLite is suitable to low- to medium-traffic websites and the one file approach of this type of DBs makes them reliable and portable. Its limitations in terms of user management and performance optimisation are unlikely to be felt in this project.

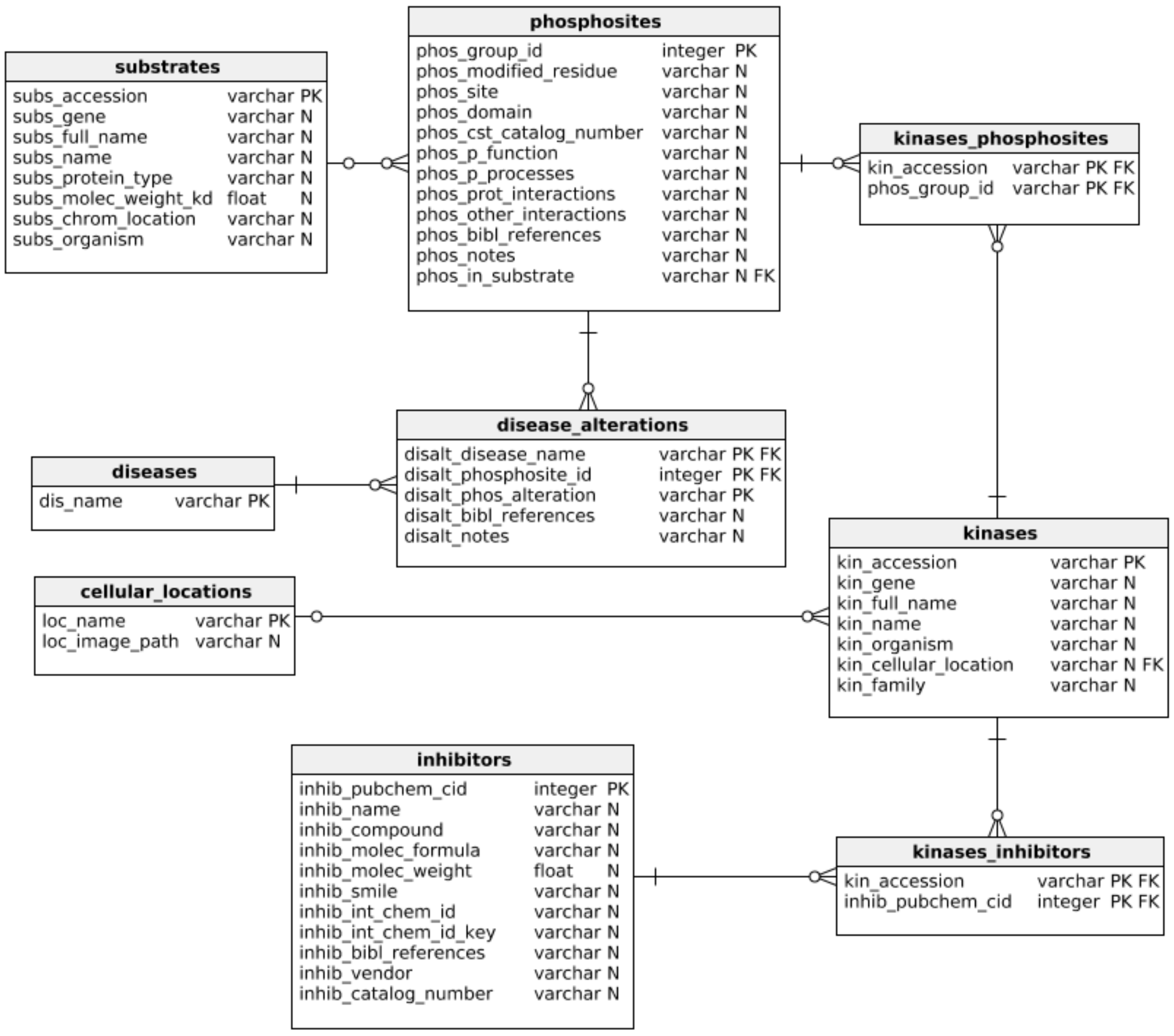
The python library SQLalchemy was employed to create and populate the DB. This makes the PhosQuest DB and WebApp more portable and allows for performance improvements in the python-SQLite interactions. With SQLalchemy, the DB can be transferred to other DB systems, with minimal changes to the `sqlalchemy_declarative.py` (table definitions; `PhosQuest_app/data_access` directory) and `db_sessions.py` (DB path and connections; `data_import_scripts` directory) scripts.

The python library `pandas` was used to parse datasets and facilitate data import. This library allows for the easy handling of large amounts of data in a time- and resource-efficient manner.

The Python libraries' versions employed in this project are specified in the project's README file. They can easily be installed using `pip install -r requirements.txt`.

Database structure

The PhosQuest database contains nine tables, two of which join tables, as outlined in the schema below:


phosphosites
phos_group_id integer PK
phos_modified_residue varchar N
phos_site varchar N
phos_domain varchar N
phos_cst_catalog_number varchar N
phos_p_function varchar N
phos_p_processes varchar N
phos_prot_interactions varchar N
phos_other_interactions varchar N
phos_bibl_references varchar N
phos_notes varchar N
phos_in_substrate varchar N FK
substrates
subs_accession varchar PK
subs_gene varchar N
subs_full_name varchar N
subs_name varchar N
subs_protein_type varchar N
subs_molec_weight_kd float N
subs_chrom_location varchar N
subs_organism varchar N
kinases_phosphosites
kin_accession varchar PK FK
phos_group_id varchar PK FK
disease_alterations
disalt_disease_name varchar PK FK
disalt_phosphosite_id integer PK FK
disalt_phos_alteration varchar PK
disalt_bibl_references varchar N
disalt_notes varchar N
diseases
dis_name varchar PK
kinases
kin_accession varchar PK
kin_gene varchar N
kin_full_name varchar N
kin_name varchar N
kin_organism varchar N
kin_cellular_location varchar N FK
kin_family varchar N
cellular_locations
loc_name varchar PK
loc_image_path varchar N
inhibitors
inhib_pubchem_cid integer PK
inhib_name varchar N
inhib_compound varchar N
inhib_molec_formula varchar N
inhib_molec_weight float N
inhib_smile varchar N
inhib_int_chem_id varchar N
inhib_int_chem_id_key varchar N
inhib_bibl_references varchar N
inhib_vendor varchar N
inhib_catalog_number varchar N
kinases_inhibitors
kin_accession varchar PK FK
inhib_pubchem_cid integer PK FK

Vertabelo

The term 'substrate' is used to refer to a protein that is a putative kinase target. 'Phosphosites' are peptides within the substrate where phosphorylation occurs, hence a substrate may have many phosphosites, which may be targetted by different kinases. The schema is defined through a SQLalchemy declarative script. Some fields were indexed to speed up DB queries:

Table	Field
`substrates`	`subs_gene`
`phosphosites`	`phos_modified_residue`
`phosphosites`	`phos_in_substrate`
`kinases`	`kin_cellular_location`
`kinases_phosphosites`	`phos_group_id`
`kinases_inhibitors`	`inhib_pubchem_cid`

Data Sources

Database Exports

All external datasets downloaded as files were saved in the `db_source_tables` directory, under the relevant sub-directory. Data on kinases, substrates, phosphosites, phosphosite regulation and disease-associated alterations was obtained from PhosphoSitePlus. Files `Disease-associated_sites.gz`, `Kinase_Substrate_Dataset.gz`, `Phosphorylation_site_dataset.gz`, and `Regulatory_sites.gz` were used to populate database tables `kinases`, `substrates`, `phosphosites`, `disease_alterations`, and `diseases`. The files were downloaded from the `Downloads` tab, `Datasets from PSP` page on 23/03/2019 (source last updated 04/03/2019). Inhibitor data was obtained from MRC Kinase Profiling Inhibitor Database as a `.csv` file on 23/03/2019 and from BindingDB as a `zip` compressed `.tsv` file on 23/03/2019 (source last updated 01/03/2019) (`Ligand-Target-Affinity Datasets` > `Only data curated from articles by BindingDB`, `BindingDB_BindingDB_Inhibition_2019m2.tsv.zip` file). Given BindingDB's file size, it could not be added in its uncompressed form to the github repo due to the latter's file size restrictions.

Application Programming Interface (API)

API functionality was dependent on the pandas module to allow handling of large datasets. The python urllib library was used for opening, reading and parsing of URLs. For our DB, we used data from UniProt and PubChem websites, which allow for searching with multiple accession numbers and output data in a data frame-compatible format for population of the DB.

To complete population of the DB, we utilised the following two website APIs:

UniProt

The UniProt website API allows for the collection of specified data using a kinase or substrate accession number.

For this - parameters are selected when performing a search - categories of information to be retrieved are indicated as columns - UniProtKB column names for programmatic access can be found here - results are returned in tab format.

To allow population of the database, we have selected the following qualifiers: `'columns': 'id,protein names,comment(SUBCELLULAR LOCATION),families,genes'` ID contains the accession number of the records to be retrieved.

In terms of functionality the code:

1. Takes parameters and encodes them in a URL format;
2. Changes to utf-8 format;
3. Requests the URL with DB record accession numbers as parameters using urllib.request;
4. Opens the respective URL and stores the data for the requested columns as a response variable;
5. Places the retrieved data into a data frame.

Based on the information retrieved, one of parameters we require to populate our database is the subcellular location. This qualifier returns multiple pieces of information relating to the subcellular location and here we only wish to retrieve the first set of information, the subcellular location. We create a separate column and using regular expression extract this information. This is also repeated for gene names where we only take the first name in the returned value.

PubChem

To access data from the PubChem website, we have utilised the PubChem REST-style version of PUG (Power User Gateway) utility which is a web interface for accessing PubChem data and services. To access the data, a URL with a specific structure is required:

The URL has three parts – input, operation, and output:

```
https://pubchem.ncbi.nlm.nih.gov/rest/pug/*input specification*/*operation specifi
cation*/*output specification*/*operation_options*
```

In our case, the selected data was obtained using the CID qualifier from the PubChem site.

This API allows a number of qualifiers to be retrieved. The full list can be accessed from the PubChem PUG-REST website.

To populate the DB, we have selected the columns:

1. IUPACName;
2. MolecularFormula;
3. MolecularWeight.

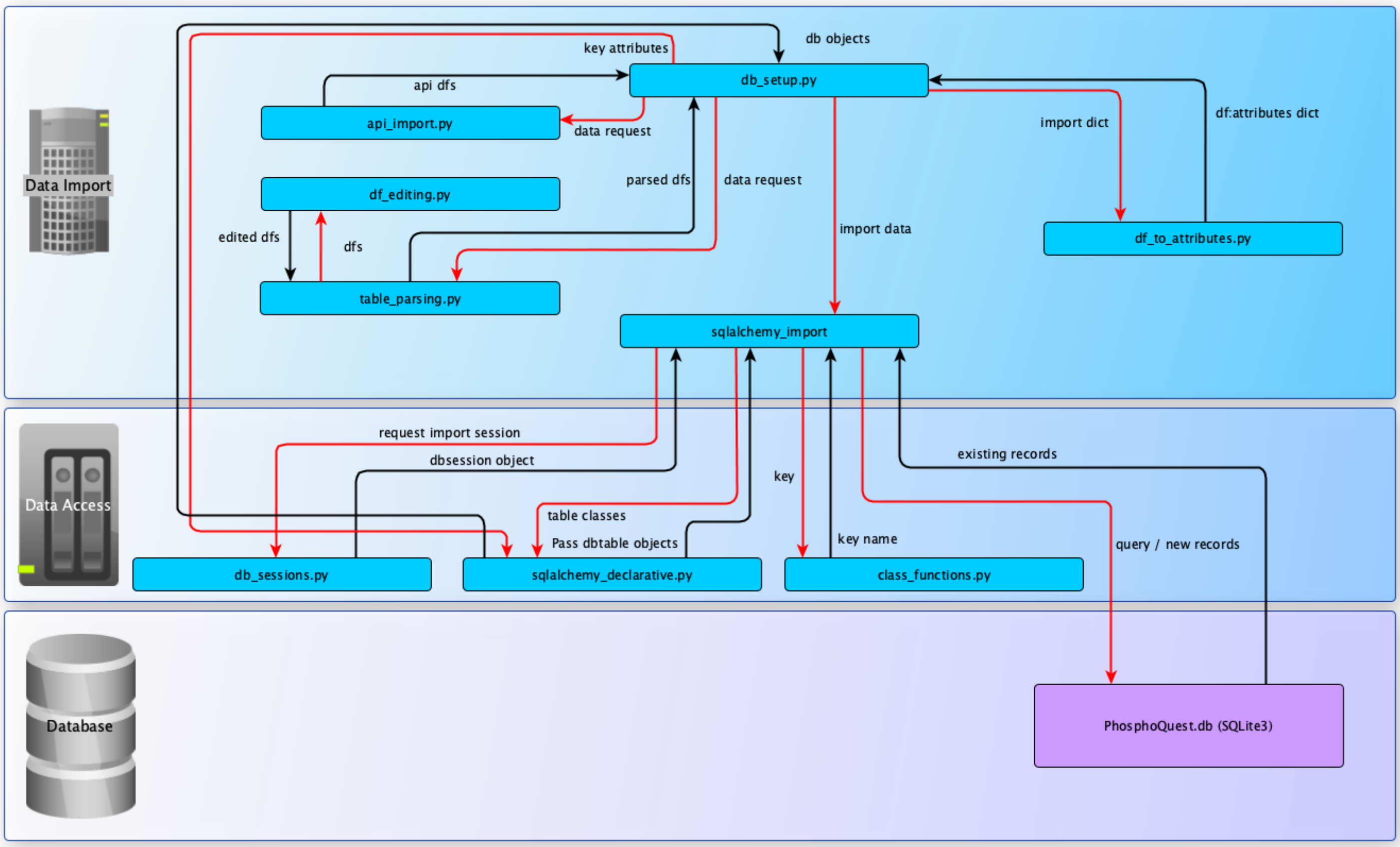
The following request was utilized to populate the `inhibitors` table:

```
results_csv = ("https://pubchem.ncbi.nlm.nih.gov/rest/pug/compound/"
               "cid/" + query_str + "/property/"
               "IUPACName,MolecularFormula,MolecularWeight/csv")
```

Where 'query_str' denotes the PubChem CID compund identifiers. The data was then converted from csv to a data frame ready for population of the database.

Database setup

To recreate the database from newly imported data, run the `db_setup.py` script which will first create the database schema if not in place already, then import the PhosphoSitePlus, MRC and BindingDB datasets, and finally obtain additional data from UniProt and PubChem using the `api_import.py` script. The script will also curate individual records found to be incorrect. The interactions between the various scripts are shown in the following diagram:



The process is outlined below:

1. Data downloaded from data sources as described in the Database Exports section above;
2. Downloaded files parsed into data frames through `table_parsing.py`, using auxiliary functions in `df_editing.py`;
3. Data frames imported into the DB through the `sqlalchemy_import.py` script (in

`data_import_scripts` directory) with auxiliary functions in `class_functions.py` and an import session from `db_sessions.py` script (both in the `data_access` directory), following the mapping of data frame column headings to SQLalchemy class objects and their attributes set out in `data_import_scripts/df_to_attributes.py`.

    - BindingDB data is filtered to include only data for inhibitors associated with kinases in the DB;

4. Specific incorrect records are curated;
5. Missing data imported for the existing records from Uniprot and PubChem APIs using the `api_imports.py` script (`data_import_scripts` directory).

Importing additional data

Additional data can be easily imported to the database by parsing a data table input into a pandas data frame and then matching the data frame column headings to the fields they are destined for in the database in the `df_to_attributes.py`. Only data for empty record fields is imported (existing data will not be over-written). To replace existing data, remove the field in question from the `df_to_attributes.py` dictionary for the undesired data source and run `db_setup.py` again.

Database limitations and potential issues

- The data imported into the DB is not fully normalised. For example, the `kin_cellular_location` and `kin_family` fields in the `kinases` table should be broken down into sub-categories across different tables to allow for automatic browse categories generation and faster querying on these fields. The same would be true for the `subs_protein_type` in the `substrates table`, for instance. To achieve this, more time would be required to fully understand the structure in these fields and adequatly parse the external data sources;
- The `diseases` and `disease_alterations` tables in the DB are not being used to show data in the web app. This is due to the difficulty in parsing their data, which was not clearly structured in the original data source. Again more time and knowledge would be needed to fully understand and parse these data;
- The fields `subs_gene` and `kin_gene` in the `substrates` and `kinases` tables were obtained from PhosphoSitePlus. Missing entries were completed using the UniProt API by selecting the most common (first listed) gene name for that protein. This could be improved if instead a single gene these fields listed all gene names associated with the protein. However this would slow down queries of the fields. In order to solve this, all genes associated with a protein accession number should be listed in a secondary table. An identical situation occurs with the protein names fields in these tables, which could be solved in a similar manner;
- Currently, the import scripts only imports data if the field for that record is currently empty. An extra argument could be passed to the import function stating whether or not each specific column should replace the existing information. In this way, it would be possible to choose preferred sources of information for each DB field;

- If the DB were to be expanded or the web app to be used by many users, the data would likely need to be migrated to a different DB format. This should be facilitated by SQLalchemy.