From Text-based Genome, Population Variations, and Transcriptome Datafiles to SQLite Database and Web Application: A Bioinformatical Study on Alfalfa

Shichen Qiao¹,*, Chen Shen²

[Abstract] In this study, a web database application with the Flask framework was developed to implement three types of queries and visualize the results over a bioinformatical dataset from Alfalfa (Medicago sativa). A backend SQLite database was constructed from genome FASTA, population variations, transcriptome, and annotation files with extensions “.fasta”, “.gff”, “.vcf”, “.annotate”, etc. Further, a supplementary command-line-based Java application was also developed for faster access to the database without direct SQL programming. Overall, Python, Java, and HTML were the main programming languages used in this application. Those scripts and the development procedures are valuable for bioinformaticians to build online databases from similar raw datasets of other species.

[GitHub Repository] https://github.com/ShichenQiao/Alfalfa_Database.git
[Web Application Demo] https://youtu.be/Djr81po206A
[Command Line Application Demo] https://youtu.be/qs05TDjWvd0

1. Introduction
The field of bioinformatics has been brought to a new era as web server technologies, internet services, and database management tools develop rapidly in recent years. Serval standard data file formats, such as the FASTA Format (.fasta), the General Feature Format (.gff), and the Variant Call Format (.vcf), are used world-widely to store raw genetic sequences, genomic annotations, and population variation data (Gourlé, n.d.). There are many other text-based annotations with different file extensions like “.annotate”, “.pfam”, and “.interpro” as well. For many species with complex chromosomal compositions, these data can be fragmented and enormous as a whole. It is inconvenient and time-consuming to do cross-file searches and analyses on these data without help from some complex online platforms or tools such as BLAST (U.S. National Library of Medicine, n.d.). However, those platforms may not be customized or convenient enough to meet every research group’s needs. Thus, integrating the genome, variation, transcriptome, and annotation data of a species into a relational database, and then utilizing a personalized website application to interact with it could be a better approach for research labs to manage their datasets.

¹ Shichen Qiao, Undergraduate Student, Department of Electrical and Computer Engineering, Department of Computer Sciences, University of Wisconsin-Madison, Madison, WI 53715, Email: sqiao6@wisc.edu
² Chen Shen, State Key Laboratory of Agrobiotechnology, College of Biological Sciences, China Agricultural University, Beijing, China, Email: shenchen94@yeah.net
* Corresponding author
In this article, a case study on Alfalfa (Medicago sativa) was conducted with MacOS 11.2 and CentOS 7. The source dataset was adopted from a journal published on Molecular Plant (Shen et al., 2020, The Chromosome-Level Genome Sequence of the Autotetraploid Alfalfa and Resequencing of Core Germplasms Provide Genomic Resources for Alfalfa Research). SQLite database was chosen because it requires fewer computer resources and it has easy connections to Java and Python programs. R, Microsoft Excel, and different text editors were also used to process data. The products of our case study include a SQLite database file (.db), a Flask-framework-based web application, and a supplementary, packaged, command-line-based Java application (.jar). Both applications have three query modes across all different-typed files in the original dataset: [A] querying about a gene ID for all related genome, transcriptome, variations, and annotation data; [B] querying about all gene IDs on file whose annotations contain some given keywords; [C] querying about all gene variation information on a certain portion of a chromosome. The web application visualizes the data through bar charts, table coloring, and categorization, and it is capable to link annotations to their source databases’ websites automatically.

2. Data Selection
The data involved in this study were adopted from genome sequencing results and transcriptome annotations on Alfalfa (Shen et al., 2020), stored as shown in Figure 1. After decompression, the FASTA, GFF, VCF, and annotation files were about 600 GB in total.

![Figure 1](image-url)
After data selection, the files shown in Figure 2 were downloaded to the Mac computer which we used primarily during the development stage. Although only 500 MB of the VCF records were kept locally, once the proper scripts were developed based on this small sample, the server would execute the same scripts and insert the complete VCF records into the database file. Further, ms.main.fasta was renamed as Ms_genome.fasta for clarity, and it was composed of the DNA sequences of 8 chromosomes and 649 contigs. Moreover, transcriptome data were stored in alltpm.txt, and the other files were annotations. A sketch map of these file formats is shown in Figure 3.
3. Software Preparation
To minimize development costs, a MacBook Pro laptop with MacOS 11.2 was chosen to be the main development environment. A CentOS 7 Linux server was used to execute some Python, Java, and SQL scripts on the large raw datafiles, and it was also used to broadcast our final web application to the Local Area Network (LAN). The essential software applications used during our development process were recorded in this section.

3.1 SQL Environment and GUI
SQLite3 is a free, server-less, self-contained, cross-platform SQL database engine (SQLite Org, n.d.). The latest version of SQLite3 is pre-installed on most Mac machines. To utilize this database engine more intuitively, a visual management tool with graphical user interfaces (GUIs), DB Browser for SQLite, was used along with the procedure (Figure 4). With this application, developers could manage the structure of their database, browse and filter data, and execute SQL scripts more conveniently (Sqlitebrowser Org, 2021).

![Figure 4. DB Browser for SQLite GUI.](image)

3.2 Visual Studio Code and Programming Languages
According to Guo (2014), Python and Java are the top two popular programming languages in US undergraduate education. We decided to use them collaboratively in our case study to take advantage of both languages’ strong points. Additionally, HTML, CSS, and JavaScript were used for the web application. As a result, we chose Visual Studio Code as our main code editor since it is capable to manage all of these languages cooperatively (Microsoft, 2016). Note that PyCharm by JetBrains S.R.O.
and Eclipse by the Eclipse Foundation are more centralized Integrated Development Environments (IDEs) that were also helpful. Specifically, Python 3.8 by Python Org., JavaSE-11 and JavaSE-1.8 by Oracle Corporation, HTML5 and CSS3 by the World Wide Web Consortium were utilized in our study. Moreover, to connect our Java applications to the SQLite database, a Java Database Connectivity (JDBC) interface was required. Maven Repository provides free SQLite JDBC on their official website, and the latest version, 3.36.0.1 was used in our development process (MvnRepository, n.d.).

### 3.3 The Flask Framework

The Flask Framework is a web application framework developed by Armin Ronacher. It manages Python web applications. Based on this framework, our HTML page and the Python application could communicate with each other, pass data inputs and query results back and forth, and broadcast the final online database website through an IP address conveniently.

### 3.4 Microsoft Excel and/or Numbers

As mentioned in the Introduction section, the annotations are stored in text-based files with different extensions. However, by observation, the annotation files are essentially Tab-Separated Values (TSV) data. Each row in an annotation file contains information about one gene ID, and that information could be divided into different “columns”.

With these characteristics, spreadsheets applications such as Microsoft Excel and Numbers are perfect choices for researchers to view and manipulate the annotations to a format that is easy to be inserted into a SQL database graphically.

In our case study, Microsoft Excel was used to convert TSV files into Comma-Separated Values (CSV) files, which enabled us to deal with annotations stored in different files with the same Python algorithm. In addition, the sorting and concatenation functions in Excel also saved a lot of time for us to generate SQL scripts.

### 4. Methods: Backend SQLite Database Development

In this section, the development procedure of our SQLite backend database was explained. There were multiple input raw data files, Python and Java scripts, intermediate data files, and SQL scripts involved during the process, but there was only one database output file (Alfalfa.db). The integration of these fragmented data files into one portable file was one of the main purposes of developing this SQL database.
4.1 Database Structure and Table Creations

The database structure was designed to make the three types of queries, as mentioned in the Introduction section, faster and simpler. By observing the datafiles, we found that the gene ID was the main connection among the datafiles shown in Figure 2. Specifically, a gene ID number could contain 13, 16, or 20 characters. For our dataset, the first 13 characters of any gene ID correspond to a specific gene of Alfalfa, so we named this part the “Primary ID”. Further, if there existed such characters, the 14th to 16th characters could only be “.01”, the 17th to 18th characters could only be “.T”, which stood for Transcript, and then followed by two digits. Thus, we named character #14 to character #20 the “extension”. Since the majority of gene IDs in the raw dataset had 20 characters, we name such gene ID as “ID_full”.

Next, we classified the datafiles into five categories. Firstly, MuXu_Last.gff contains 580,707 rows of 9-column GFF entries. Using data in the seqname, start, and end columns, we could chop out the DNA sequence for each GFF record from Ms_genome.fasta and add it to a “DNA” column following the existing “attribute” column (The Ensembl Project, n.d.). We then give each row a unique_id as Primary Key (PK), and we could obtain Table gff as shown in Section 10.1 on Line 58 through 71. We also added a SQL Table called “ID_to_uniqueid” to map each primary ID to its contiguous block of unique_ids, and a Table called sqlite_sequence was created by our environment automatically at this point. Secondly, the go, kegg_pathway, pathway, pfam, interpro, and pros.fasta datafiles were classified as “info” datafiles. Specifically, each entry in these files started with a 20-character gene ID and then followed by a various number of data cells. Thus, we used ID_full as PKs and created Table go_info, interpro_info, kegg_info, pathway_info, pfam_info, and pros_info as shown in Section 10.1 on Line 73 through 114. In addition, transcriptome data stored in alltpm.txt were planned to be transferred into a 48,807 row by 38 column SQL Table directly, as shown on Line 9 through 48. Moreover, we had MuXu_Last.annotate storing gene annotations from other online databases of *Medicago Truncatula*. This file was opened in Microsoft Excel as Tab Separated Values (TSV), as shown in Figure 5, and we took out column A (ID_full), D (db_name), F (info1), and M (info2) to create Table annotate_info for keywords queries, as shown in Section 10.1 on Line 50 through 56. Finally, we observed the 500 MB VCF sample and found that it was structurally a table with 167 columns. Thus, we placed a Primary Key column called “vcf_ID” in front of the original columns and created Table vcf as shown on Line 118 through 288.
4.2 Pre-processing Annotation and FASTA Datafiles with Python

After table creations, the raw data needed to be pre-processed before insertions into our database through SQL scripts. In this section, methods of pre-processing FASTA and annotation files using Python would be elaborated.

We started with the go, kegg_pathway, pathway, pfam, and interpro files because they could be managed with very similar scripts. Observing the interpro annotation file, we noticed that no matter there were how many interpro IDs for each gene ID, they were all concatenated together using "|" as the separator. This could easily solve the problem that, in the other annotation files, the number of annotations attached to an ID_full was not unified. Thus, for the other four files, the following Python script was used to concatenate the annotations for each ID_full using semicolons as the separator:

```python
1. import csv
2. with open("go_code.csv", 'r') as fp:
3.     reader = csv.reader(fp)
4.     for x in reader:
5.         temp = ""
6.         for letter in x[0]:
7.             if letter == ";":
8.                 temp = temp + letter
9.             else:
10.                temp = temp + letter
11.         temp = temp.rstrip("")
12.     content.writerow([temp])
13.     fp.close()
14.     rp.close()
```

In addition, MuXu_Last.pros.fasta contained data with similar properties as well. Each FASTA entries started with a “>” followed by an ID_full, and then a protein
sequence was on its following line. Thus, this file could be converted into a 2-column CSV table of key-value pairs, similar to the five files pre-processed above, by running the following Python script:

```python
1. import csv
2. def readFa(fa):
3.     with open(fa,'r') as FA:
4.         seqName,seq="",""
5.     while 1:
6.         line=FA.readline()
7.         line=line.strip('
')
8.         if (line.startswith('>') or not line) and seqName:
9.             yield((seqName,seq))
10.        if line.startswith('> '):
11.            seqName = line[1:]
12.            seq=""
13.        else:
14.            seq+=line
15.        if not line: break
16.     with open('protein_sequence.csv', 'w') as fp:
17.         content = csv.writer(fp)
18.     fa = "./MuXu_Last.pros.fasta"
19.     for seqName, seq in readFa(fa):
20.         content.writerow([seqName, seq])
21.     fp.close()
```

### 4.3 INSERT Pre-processed Data to SQLite Tables

After pre-processing, all files in our original dataset shown in Figure 2 except Ms_genome.fasta and the VCF file were ready to be converted into SQL INSERT statements. Those treated CSVs and TSVs could be opened in Microsoft Excel as spreadsheets formats and used to generate SQL scripts. For instance, when we were processing the GFF datafile, we changed the “.gff” extension to “.txt”, and then opened it with Excel using Tab as the delimiter. Next, the data were sorted by the ASCII order of the attribute section, which made sure that GFF entries belong to the same gene ID were clustered. Finally, the “unique_id” column was inserted into the spreadsheet, giving each row of GFF entry a unique tag as we planned (Figure 6).
Figure 6. Part of MuXu_Last.gff opened in Excel with unique_id added.

At this point, all data in the GFF file had been prepared in a table format for insertion into the database. A build-in function of Excel called “CONCATENATE” was used at the K1 data cell of the spreadsheet shown in Figure 6:

1. `=CONCATENATE("INSERT INTO gff(unique_id, seqid, type, start, end, score, strand, phase, attributes) VALUES("&A1&","&B1&","&C1&","&D1&","&E1&","&F1&","&G1&","&H1&","&I1&","&J1&")");`

An example of the resulting SQL command is:

1. `INSERT INTO gff(unique_id, seqid, type, start, end, score, strand, phase, attributes) VALUES("1", "Chr1", "maker", "gene", "146833", "150094", "+", ".", ".", ID=MsG1A1000001.01;Name=MsG1A1000001.01);`

Later, the whole K column was selected, and a shortcut key, Control plus Enter, was used to implement the equation to each row and generate the complete insertion SQL script automatically. Copying the entire column K to a text file with the extension “.sql” and executing it in DB Browser for SQLite gave us Table gff as shown in Figure 7.
For the pre-processed CSV files, they could be opened in Excel with comma as the delimiter, and then similar CONCATENATE functions could be applied. Notice that not all characters could be handled directly using this procedure. For example, while creating Table annotate_info, we found that 15 Excel-generated SQL commands had syntax errors because the character "-" was not supported in INSERT instructions. One of these invalid commands was:

1. \texttt{INSERT INTO annotate\_info(ID\_full,db\_name,info1,info2) VALUES(\textquotesingle MsG1A1000031.01.T01\textquotesingle, \textquotesingle ProSitePatterns\textquotesingle, \textquotesingle Lipolytic enzymes \textquotesingle \textquotesingle G-D-S-L\textquotesingle family, serine active site\textquotesingle,"Lipase, GDSL, active site\textquotesingle);}

Since there were only 15 exceptions, we added these entries manually to Table annotate\_info via DB Browser for SQLite. If there were much more exceptions, replacing these characters might be necessary.

Moreover, when the source datafile contained too many columns, the convenience of using Excel would be largely reduced. In such cases, using Python scripts to convert the text-based file directly to SQL INSERT statements could be a better option. For example, alltmp.txt was processed by the following Python program:

1. \texttt{import csv}
2. \texttt{with open\textquotesingle alltmp.csv\textquotesingle, \textquotesingle r\textquotesingle\texttt{) as fp:}
3. \texttt{reader = csv.reader(fp)}
4. \texttt{rf = open\textquotesingle alltmp.sql\textquotesingle, \textquotesingle w\textquotesingle)
6. for row in reader:
7.     values ="
8. for x in row:
9.     values +="\n + \n + \n,
10.    values = values[0 : len(values) - 1]
11.    sql = "INSERT INTO alltpm(GeneID, "150_trueleaf_rep1", "150_xia_rep1", "150_ziye_rep1", fennie_rep1, flower_bud_rep1, flower_rep2, flower_rep3, flower_rep4, flower_rep5, flower_rep6, flower_rep7, flower_rep8, germ_seed_rep1, guojia_rep1, lateral_root_rep1, leaf_bud_rep1, leaf_bud_rep2, leaf_bud_rep3, main_root_rep1, mature_leaf_rep1, mature_leaf_rep2, mature_leaf_rep3, MS_trueleaf_rep1, MS_xia_rep1, MS_ziye_rep1, nod_21_rep1, nod_28_rep1, seed_rep1, stem_bran_rep1, stem_rep1, yige_rep1, yige_rep2, yige_rep3, young_leaf_rep1, young_leaf_rep2, young_leaf_rep3) VALUES(" + values + ");"
12. rf.write(sql)
13. fp.close()
14. rf.close()
import csv

with open('uniqueid_ID_primary.csv', 'r') as fp:
    reader = csv.reader(fp)

rp = open('linkage_ID_to_uniqueid.csv', 'w')

content = csv.writer(rp)

temp = -1
start = 1
end = 1

for x in reader:
    if x[0] == 1:
        name = x[1]
    if x[1] != temp:
        if temp != -1:
            print(name, 'starts from', start, 'ends at', end - 1)
        content.writerow([name, start, end - 1])
    name = x[1]
    start = int(x[0])
    temp = x[1]
    end = end + 1

content.writerow([x[1], start, x[0]])

fp.close()

rp.close()

The for-statement traverses through each row of the CSV file, and if the name (primary ID) changes between two consecutive rows, the starting and ending unique_ids would be recorded in the output CSV file.

Using a similar formula, these data were inserted into Table ID_to_uniqueid by executing an Excel-generated SQL script using the following function:

```
=CONCATENATE("INSERT INTO ID_to_uniqueid(ID, start, end) VALUES("&A1&", ",&B1&", ",&C1&")");"
```

Notice that when the data being inserted does not contain double quotation marks or apostrophes, using "&A1&" and ",&B1&", ",&C1&" in the function above would generate the same SQL commands.

Further, for each GFF entry regardless of its type, the substring obtained from the corresponding FASTA entry stored in Ms_genome.fasta between (closed interval) the indexes specified in the “start” and “end” columns in Table gff should be the detailed DNA sequence matched to that GFF row. Thus, Ms_genome.fasta was chopped 580,707 times to obtain the SQL scripts to fill the “DNA” column of Table gff.

However, keeping the entire FASTA file open for these many operations was not efficient. Thus, the following Python script was used to divide Ms_genome.fasta into 657 text files. Each file was named as a seqid followed by “.txt”, and the content was the complete DNA sequence attached to that seqid (the readFa function was defined in Section 4.2).
Next, the “unique_id”, “seqid”, “start”, and “end” columns in Table gff was exported as a CSV file called “start_end_raw.csv” from Table gff, and the following Python script was used to truncate the sequences as described above and generated 580,707 SQL UPDATE commands:

```python
import csv
with open('start_end_raw.csv', 'r') as fp:
    reader = csv.reader(fp)
    rf = open('sequences.sql', 'w')
    f = open('Chr1.txt', 'r')
    for x in reader:
        unique_id = x[0]
        seqName = x[1]
        start = x[2]
        end = x[3]
        fileName = seqName + '.txt'
        f = open(fileName, 'r')
        data = f.read()
        f.close()
        seq = data[int(start) - 1:int(end)]
        sql = "UPDATE gff SET DNA = " + seq + " WHERE unique_id = " + unique_id + "\n"
        rf.write(sql)
fp.close()
rf.close()
```

After executing sequences.sql, Table gff was finalized (Figure 9).

![Figure 9. Finalized Table gff with DNA sequences.](image-url)
4.5 Improving Table annotate_info
So far, all data except the VCF file had been added into the SQLite database. However, 81,266 duplicated rows were found in the Table annotate_info. This was caused by the replications of annotations on the same gene in the raw “.annotate” file. To eliminate the redundant copies, only the row with the smallest row number was kept for each cluster of rows with the same contents after running this SQL command:

1. DELETE FROM annotate_info WHERE rowid NOT IN (SELECT min(rowid) FROM annotate_info GROUP BY I D_full, db_name, info1, info2);

Further, to make frontend development easier, all empty data entries (shown as “” in INSERT statements) was replaced by the NULL character by the following SQL commands:

1. UPDATE annotate_info SET info1 = NULL WHERE info1 = ""
2. UPDATE annotate_info SET info2 = NULL WHERE info2 = ""

4.6 Building Table vcf
The last type of data was the gene variation records stored in the massive VCF files. In this section, the approaches we had taken to transfer VCF data into our SQLite database were discussed.

The first step was analyzing how VCF data were stored. By standards, there is a single row in the VCF file that begins with only one “#”, and all column names are listed on that row. The rows above are file headers that begin with “##” on each row, and each row below describes gene variations at a specific position. That row with column names was taken out, put into a file editor, and stored as a TSV file called “column_names.txt” (Figure 10).

![Figure 10. column_names.txt.](column_names.txt)

Next, we decided to give each variation record a unique number, because a Primary Key was needed to improve searching efficiency. Thus, a similar variable like we
used for Table gff was utilized again. To be more specific, a column of consecutive natural numbers, called “vcf_ID”, was added to each row of Table vcf. The beginning and ending vcf_ID of each chromosome would be recorded by the frontend applications to distinguish different chromosomes.

With all column names and properties decided, the following Python script was executed on this text file to generate the CREATE statement in SQL:

```python
with open('create_vcf.sql', 'w') as fp:
    sf = open('column_names.txt', 'r')
    dataIn = sf.read()
    columns = dataIn.split('
')
    sf.close()
    fp.write("CREATE TABLE vcf (vcf_ID INTEGER NOT NULL UNIQUE, ")
    # remember to change column POS to INTEGER from TEXT
    for x in columns:
        fp.write("" + x + ""
    fp.write(" PRIMARY KEY("vcf_ID") AUTOINCREMENT)\n);")
```

The scripts above unified all columns stored in column_names.txt to be TEXT data for simplicity, but it was necessary to make the “POS” column contain INTEGERS. The next step was filtering out the gene variation data in the VCF file (as shown in Figure 11), and converting it into a more human-readable way. By our design requirements, we were only interested in the actual nucleotides at each position (POS). In the GFF file, “.” means the variation data at the corresponding position in a specific breed of Alfalfa is not applicable. If for a certain breed, gene variation data is available, that data would be in a format like “0/1:2:3:28:67,0,28”. The number right before and right after the “/” character is most important to our database. If a number was 0, it means that the gene at that specific position was the same as the “REF” value on the same row; if a number was 1, it means the gene at that position is a variant same as the first variation listed in column “ALT”; if the number was 2, it matches to the second variation in column “ALT”; etc.

Figure 11. Part of raw VCF data.
Manually looking back and forth between the numbers and the “REF” column was not efficient. Thus, we replaced all numbers with the actual nucleotides they represent, and generated SQL INSERT statements for the VCF data by running the Java application shown in Section 10.2. Finally, executing the generated SQL script gave us Table vcf in our database file (Figure 12).

Note that the same process was applied to the concatenation of all 8 original VCF files, which was about 592 GB, on our CentOS server to get the complete Table VCF.

### 4.7 Constructing vcf_jump_table for Faster Access

In our case, there were about 150 million rows of VCF data in the complete dataset from Alfalfa. However, our query inputs were supposed to be seqid and POS values, instead of vcf_ID, the Primary Key. This could largely harm the query efficiency because SQLite would go through all POS values within the Table for every VCF query. Generally speaking, for integer data columns like POS, adding an index to that column could improve query efficiency. Yet, in our case, although the POS values were monotonically increasing for each chromosome, the same POS value could appear multiple times at a different or even the same chromosome. For instance, within the first 500 MB of VCF data from Chromosome #1, there were 5,660 rows with POS values appeared more than once (as shown in Figure 13). This made indexing very complicated.
Thus, a jump table was created as follows:

```sql
CREATE TABLE "vcf_jump_table"(
  "vcf_ID" INT NOT NULL UNIQUE,
  "POS" INT NOT NULL,
  PRIMARY KEY("vcf_ID")
);

INSERT INTO vcf_jump_table(vcf_ID, POS) SELECT vcf_ID, POS FROM vcf WHERE vcf_ID % 10000 = 0;
```

This way, for every 10,000 vcf_IDs, the first POS value would be stored in Table vcf_jump_table as shown in **Figure 14**. Searching the POS value being queried in this jump table could help the frontend application to limit the query range of vcf_ID from an entire chromosome to a small segment of it. The detailed query commands would be elaborated in Section 5.3.
5. Methods: Frontend Flask Web Application

With the database file developed so far, we could move on to frontend design. In this section, we would refer to our three query modes as ID Query, Keywords Query, and VCF Query. The frontend web application should be capable to execute searches with user inputs from the website, giving correct and visualized query results, and handle improper inputs with detailed instructions. These functions were achieved under an isolated Python environment created by a Python package called “virtualenv”. The web application itself was developed under the Flask web framework, which was based on Jinja Template Engine. Jinja enabled the Python program to get and pass data to the HTML page and render proper query results on the web pages shown to the users (Ronacher, n.d.).
5.1 Overall Look on the Web Project

As shown in Figure 15, the final project folder of the frontend web application contains a README document, the main Python program called “app.py”, a virtual environment folder, a static resource folder, and an HTML templates folder. Looking deeper into our folder levels, there were 8 txt files in the DNA folder containing the complete sequences of the 8 chromosomes of Alfalfa, which was copied from the intermediate datafiles created in Section 4.4. The backend database, Alfalfa.db, and two SQL scripts were stored in the folder “sql”. Those two scripts were used to get the marginal vcf_ID values for each chromosome’s VCF records. Moreover, the CSS file contained the format settings of our website and the three HTML templates were used interactively with the Python program to render the GUI of our database on a website. The complete scripts in the Python, CSS, and HTML files were shared in the Appendix section, and on our GitHub repository.
5.2 SQL Query Statements

For ID Queries, the frontend application was required to SELECT all information about a gene ID from Table annotate_info, pros_info, gff, alltpm, and the other five Tables containing annotations from other databases. Retrieving annotate_info, pros_info, and alltpm data were straightforward. The following SQL SELECT commands could be used to take out all rows in those three Tables associated with the given ID (using MsG1A10000001 as an example):

1. `SELECT * FROM annotate_info WHERE ID_full LIKE "MsG1A10000001%";`
2. `SELECT * FROM pros_info WHERE ID_full LIKE "MsG1A10000001%";`
3. `SELECT * FROM alltpm WHERE GeneID = "MsG1A10000001.01";`

To make querying GFF information more efficient, an implicit JOIN instruction was utilized:

1. `SELECT * FROM gff AS G, ID_to_uniqueid AS I WHERE I.ID = "MsG1A10000001" AND G.unique_id >= I.start AND G.unique_id <= I.end;`

Noticed that the “ID” column in Table ID_to_uniqueid only contained the 13-character primary ID, so that we used “I.ID = ” here without the “%” wildcard. This way, the query speed could be improved. Moreover, explicit LEFT OUTER JOIN was used to find annotations from other databases:

1. `SELECT pros_info.ID_full, pros_info.unique_id, kegg_info.kegg, pathway_info.pathway, go_info.go, interpro_info.interpro, pfam_info.pfam FROM pros_info
2. LEFT OUTER JOIN go_info ON pros_info.ID_full = go_info.ID_full
3. LEFT OUTER JOIN interpro_info ON pros_info.ID_full = interpro_info.ID_full
4. LEFT OUTER JOIN kegg_info ON pros_info.ID_full = kegg_info.ID_full
5. LEFT OUTER JOIN pathway_info ON pros_info.ID_full = pathway_info.ID_full
6. LEFT OUTER JOIN pfam_info ON pros_info.ID_full = pfam_info.ID_full
7. WHERE pros_info.ID_full LIKE "MsG1A10000001.01.T01";`

This SELECT statement could give us the correct result because all possible ID_full was stored in Table pros_info, and if some of the Tables did not contain the provided gene ID, the NULL character would be used as a placeholder because of the property of OUTER JOINs. If the gene ID was not included in any of these five Table, the frontend applications could detect those NULL characters and hide them respectively.

The Keywords Query mode could be restated as follows: if a row of data in Table annotate_info contains the entered keywords as a sub-string of the data string stored in column “info1” and/or column “info2”, the entire row should be returned. Fuzzy searches like this could be managed by the wildcard symbol, %, in SQL. The following SELECT statement was created to achieve these goals (using keywords “HSF” as an example):
Finally, as we planned in Section 4.7, we use two cascading searches for VCF Queries. Since HTML rendering could be time-consuming, we restricted the query range of POS to be between 1 base pair (bp) to 3,000bp. This could be easily changed by modifying the if statement on Line 257 of app.py. Additionally, we were required to know the first and last vcf_ID of each different chromosome. These values were obtained by running get_max_vcfID.sql:

1. `SELECT MAX(vcfd_ID) FROM vcf;`

and get_min_vcfIDs.sql:

1. `SELECT CHROM, min(vcfd_ID) FROM vcf WHERE CHROM = "Chr1";
2. `SELECT CHROM, min(vcfd_ID) FROM vcf WHERE CHROM = "Chr2";
3. `SELECT CHROM, min(vcfd_ID) FROM vcf WHERE CHROM = "Chr3";
4. `SELECT CHROM, min(vcfd_ID) FROM vcf WHERE CHROM = "Chr4";
5. `SELECT CHROM, min(vcfd_ID) FROM vcf WHERE CHROM = "Chr5";
6. `SELECT CHROM, min(vcfd_ID) FROM vcf WHERE CHROM = "Chr6";
7. `SELECT CHROM, min(vcfd_ID) FROM vcf WHERE CHROM = "Chr7";
8. `SELECT CHROM, min(vcfd_ID) FROM vcf WHERE CHROM = "Chr8";`

over the complete database file with VCF data from all of the 8 chromosomes on our CentOS server. These SQL query results were stored as pre-defined values in our frontend code on Line 264 and 265 in app.py. When an actual VCF Query happens, our frontend application first executes the following two SQL queries on Table vcf_jump_table:

1. `SELECT * FROM vcf_jump_table WHERE vcf_ID BETWEEN <vcfd_ID_low> AND <vcfd_ID_high> AND POS <= <input_low> ORDER BY vcf_ID DESC LIMIT 1;`
2. `SELECT * FROM vcf_jump_table WHERE vcf_ID BETWEEN <vcfd_ID_low> AND <vcfd_ID_high> AND POS >= <input_high> LIMIT 1;`

<vcfd_ID_low> and <vcfd_ID_high> were obtained from the pre-defined array, matching to the specified seqid selected by the user, and <input_low> and <input_high> were the specified POS range. Since we had limited the POS range to be less than or equals to 3,000bp, these two queries were likely to return a query range of vcf_ID less than 20,000 rows. This narrowed a VCF Query task from one search over about 20-millions of rows down to two queries over tens of thousands of rows per query. It also stops the application from going deeper on unrelated VCF records to look for POS values within the specified range.

Finally, we used the value returned above, vcf_ID_lower_bound and vcf_ID_upper_bound to pull out the section of rows on Table vcf whose POS value were between <input_low> and <input_high>: 
1. SELECT * FROM vcf WHERE vcf_ID BETWEEN <vcf_ID_lower_bound> AND <vcf_ID_upper_bound> AND POS BETWEEN <input_low> AND <input_high>;

5.3 Frontend Python Application

The Python application, app.py, was responsible to connect to the backend database, execute queries according to the HTML inputs, and render HTML templates with the visualized query results. There are four URL routes controlled by this program: “/”, “/ID/”, “/annotation/”, and “/VCF/”.

To begin with, the home route, “/”, renders the default home page with user inputs placeholders, as defined in Section 10.5 on Line 9 through 57. Whenever the user refreshes the page or clicks the reset button, the application should be redirected to this route and render this page as shown in Figure 16.

![Figure 16. Web application home page.](image)

Secondly, route “/ID/” manages ID Queries. The only input is obtained from the text box on the top of the home page. As mentioned before, a valid gene ID could only have 13, 16, or 20 characters for our dataset. Thus, any input string with a different length is rejected and the error.html page, fully defined in Section 10.5, should be rendered with a proper error message. If the route is not returned at Line 33 of app.py as defined in Section 10.3, it would be connected to Alfalfa.db using the Python package called “sqlite3”, and then executes queries over all SQL Tables except Table vcf as planned in Section 5.2. There are flag variables such as show_annotate and have_info for the HTML templates to hide “None” results for different SQL queries. Note that Table pros_info is searched first on Line 38 through 44 because all valid ID_full were stored in this Table. If Table pros_info does not contain the gene ID from the HTML user input, the error page should be rendered and inform...
the user that there is no information found about the given gene ID. Further, while querying from Table go_info, interpro_info, kegg_info, pathway_info, and pfam_info, the concatenated data are divided to list objects, so that each annotation entry could be correctly hyperlinked to their source pages from other databases’ websites. Moreover, while querying GFF data, many resulting GFF entries’ “strand” value is “-”. In such cases, the reverse complement DNA sequence in the “DNA” column should be printed to the web page. On the other hand, if the “strand” value is “+”, directly printing the DNA sequence stored in the backend database would be sufficient. To achieve this, a helper method, get_conjugate, was defined on Line 6 through 19 and utilized on Line 147 through 152. Next, all CDS sequences in the GFF results would be concatenated together on Line 154 through 166 for better user references. Finally, expression quantity data about the queried Primary ID stored in Table alltpm are stored in a Python list object, as shown on Line 168 through 174. The third route, “/annotation/”, executes queries over Table annotate_info. Note that there could be more than one annotation entry attached to the same 20-character gene ID because those annotations could come from different source databases of Medicago truncatula. Thus, a rowspan attribute is calculated for each row in the query results, and added to the end of each tuple object in ans_list, as shown on Line 208 through 236. For each ID_full in the result set, the top row has a rowspan value equals to the number of all rows attached to that ID_full, and the rest of rows’ rowspan values were set to 0. This way, the HTML page could only print each unique ID_full only once. Finally, route “/VCF/” returns the proper section of Table vcf on the seqid specified in the drop-down menu, as shown in Figure 16, between the specified POS range. The complete DNA sequence being queried would be extracted from the proper txt file stored in the folder “DNA”, using a similar method as discussed in Section 4.4.

5.4 Frontend HTML Templates and Cascading Style Sheets (CSS)
The HTML templates and the CSS file were responsible to interact with users via the website, collect user inputs, format and visualize query results passed back from the Python program, and add hyperlinks both internally and externally to other existing online databases. Since the Jinja template engine is extensible, we could inherit HTML codes and settings from other parental HTML templates. In our case study, both the home.html and error.html templates inherited codes from base.html, as shown in Section 10.4 through 10.6. Since error.html renders the same user interface, with only an additional error message area, as home.html does in its reset state as shown in Figure 16, only the design of home.html with main.css is elaborated in this section. Screenshots of sample queries would be shared below in Section 6.
Line 9 through Line 108 of home.html are responsible to render the home page, shown placeholders at reset, and echo the most recent query inputs. Note that echoing text inputs with empty space characters and remembering dropdown menu selection might require help from JavaScript to process the input value passed back from the main Python application, as shown on Line 68 through 79 and Line 86 through 94.

Further, for query mode A, ID Query, we visualized our query results into four separate tables and a bar chart. Those tables were put into a CSS class called “container” to force the query result window to stay at an acceptable height. Specifically, Line 114 through 171 in home.html could create a table of hyperlinks. Any KEGG, Pathway, GO, Interpro, and Pfam annotation on the gene being queried would be shown with clickable links to their source databases’ websites. A second resulting table would contain annotation keywords from those outside databases. Below the second table, a bar chart, powered by Apache Echarts (n.d.), should be rendered to visualize the expression quantity as defined on Line 199 through 243 in home.html. Moreover, a third table would print the complete protein sequence and CDS sequence for each queried ID_full, and the last table should contain the complete GFF records related to the given gene ID. Note that if any of these tables have no meaningful content, the table headers would be hidden accordingly to the flag variables passed from app.py. To make the tables resulting from ID Queries more human-readable, all DNA and protein sequences were defined to be in a “seq” class. The CSS file had regulated objects in this class to shown its text content completely, as defined on Line 36 through 40 so that users could have a better impression of the length and characteristics of the sequences they found. Additionally, we had taken advantage of the “scroll” mode for “x-overflow” and “y-overflow” in main.css to prevent long text data from occupying nearby data cells.

In addition, in the Keywords Query mode, Line 298 through 323 in home.html was used to format the resulting table according to the rowspan attribute passed from app.py. For each unique ID_full in the result set, internal links to its ID Query result were managed by hidden input clauses on Line 312.

Finally, VCF Queries were taken care of by Line 329 through 578. The input parameters and their corresponding complete DNA segment were designed to be shown in a separated table defined on Line 332 through 354. The resulting VCF table would contain all VCF data within the specified range, and data cell coloring was implemented to improve data visualization. Specifically, the reference (REF) gene type was colored green, the heterozygous gene type with a REF strand was colored orange, the heterozygous gene type with no REF strand was colored yellow, and the homozygous mutation type was colored pink. This way, researchers could quickly detect the genotypes and patterns they are looking for. Note that the resulting VCF
table was given a style “user-select: all” on Line 359. This is critical to make copying the entire resulting set and pasting to Excel spreadsheets more convenient.

6. Results
Overall, the entire project flowed as shown in Figure 17. The resulting web application perfectly satisfied our design requirements. The three query modes, ID Query, Keywords Query, and VCF Query were all managed properly with good preciseness, intuitive visualization, and detailed exception handling. The complete project folder was shown in Figure 15, and the default home page was shown in Figure 16 in the previous section. In this section, sample queries are shown as the result of our study on Alfalfa. A demo video of our web application could also be found at: https://youtu.be/Djr81po206A.

Figure 17. Flow Chart of the entire project.
To begin with, Figure 18 through Figure 21 present the result of a 16-character ID Query. All hyperlinks to external online databases work as expected, and all information related to the given ID in our raw dataset are retrieved precisely.

The resulting tables are all flexible. When there are more than one sequences related to a given gene ID with 13 or 16 characters, the complete CDS and protein sequences table would incorporate all of those records (Figure 23). On the other hand, any Null or None characters in the query result would be hidden as expected (Figure 22).
Moreover, if a 20-character gene ID is searched, only that specified sequence is shown in the resulting table. Any other sequences or GFF entries from the same primary gene ID are hidden, as shown in Figure 24.

Secondly, for Keywords Queries, the internal links to ID Query results and the combined data cell formats are rendered as we planned (Figure 25).
Moreover, in VCF Query mode, the expected portion of the complete VCF table is returned (Figure 26). The nucleotides at each POS are visualized precisely, and onetaping copying from the resulting VCF table to Excel spreadsheets is supported.

Finally, if any user inputs were invalid, or if there was no result found, error.html was rendered with the proper error message, as shown in Figure 27 and Figure 28.
Lastly, the static resource folder of our final product is shown in Figure 29. Our backend SQLite database file not only made the original dataset more structured and portable, but also compressed the dataset from near 600 GB down to 114 GB.

```
[root@utlab alfalfa_database_bak]# tree -h ./static/
./static/  
|-- [ 30] css
 |  `-- [ 73] main.css
|-- [176] DNA
 |  |-- [ 98] Chr1.txt
 |  |-- [ 81] Chr2.txt
 |  |-- [ 99] Chr3.txt
 |  |-- [ 88] Chr4.txt
 |  `-- [165] Chr5.txt
 |     |-- [169] Chr6.txt
 |     |-- [ 91] Chr7.txt
 |     `-- [ 87] Chr8.txt
|-- [144] sql
 |  |-- [7.6] Alfalfa_Database.jar
 |  |-- [114] Alfalfa.db
 |  |-- [28] get_max_vcfID.sql
 |  |-- [ 45] get_min_vcfIDs.sql

3 directories, 13 files
```

Figure 29. Resulting static resource folder.

In all, our Alfalfa Database web application is the first comprehensive, online, bioinformatical database of *Medicago sativa*. Although the product is still under testing on our local area network, the development process and scripts used along
the procedure have referential significance to develop online databases of other species from similar text-based datasets.

7. Discussions
Although all of our Python, SQL, and Java scripts worked smoothly while constructing the database’s backend and deploying the web frontend, we have also discovered some potential improvements.

7.1 Basic Improvements on Our Procedure
Firstly, the process of converting TSV annotations to SQL INSERT scripts could be improved. In our case study, we changed the raw data files’ extension to “.txt”, opened them in Excel spreadsheets, and used the CONCATENATE function in Excel to generates the SQL instructions. However, when the annotation file is large enough, Excel may crash when applying the formula to millions of data cells at the same time on a personal computer. In this case, using a Python script to process the TSVs and generate SQL commands directly could be a more stable way. Drabas (2016) has provided a convenient way to read and write TSV files with a Python package called “pandas”. With the help from that package, scripts similar to what we used to generate the UPDATE statements in Section 4.4 could be used to process the raw data.

In addition, sorting some of the raw data could improve the processing time. For instance, when we truncated the main FASTA file into DNA sequences, if Table gff was sorted by seqid, it would be no longer necessary to open and close text files for every GFF entry. Instead, the text files could be opened only once in a predefined order, such as an alphabetical one, and a file would only be closed once after processing the entire continuous block of GFF entries from that corresponding DNA sequence. This way, thousands of unnecessary file-open and file-close operations could be prevented.

7.2 R and RStudio
In addition to Python, R is another popular programming language for statistical analysis developed by the R Foundation, and RStudio is a widely used IDE for R (The R Foundation, n.d.; RStudio PBC, 2021). Some R packages, such as Biostrings (Figure 30) and vcfR (Figure 31) could give researchers a statistical view on the massive raw Genome Sequencing and Transcriptome data (GFF and FASTA files). In this study, these statistics were used to check the programs’ correctness. For future studies and developments, vcfR could be a powerful tool to manipulate, visualize, and increase the readability of the information stored in VCF files.
After we deployed the complete Alfalfa Database web application to our CentOS server for user testing, we noticed that the HTML rendering time was still a problem. For some users that prefer access speed to the genome or transcriptome data over good visualizations, waiting for the website to construct hyperlinks and format the tables could be unnecessary. Thus, a supplementary command-line-based Java frontend application has been developed for faster access to our database file, as shown in Section 10.8. Note that for VCF Queries in this application, the POS range was not restricted, so that users could work with the VCF data more productively. Some sample queries were recorded in this video: https://youtu.be/qs05TDjWvd0.

This Java application was mainly based on a JDBC package, sqlite-jdbc-3.36.0.1.jar, and the SQL query statements were very similar to what was utilized in the web application. Those queries were executed via the stat.executeQuery(<SQL SELECT statements>) method on Line 95, 111, 141, 155, 196, and 272. After the query results were returned to a ResultSet object called rs, rs.next() would return true when there were more rows of unprocessed results. In this case, rs.getString(<column name>) was called to read out and print the proper searching result to the console window. To improve the readability of query results on the command line, variable cnt was used repeatedly throughout different queries to count how many meaningful rows of data had been returned. Many other formatting details and strategies to eliminate NULL results were also applied to improve user experience.

> seqdata
BStringSet object of length 657:

| width | seq   | names         |
|-------|-------|---------------|
| [1]   | 100287719 TTTACTCGTTTCTAAGTC...ATAAGCATAAGATGCAA Chr1 |
| [2]   | 84774831 TCGCITCAAGAGGTAAA...ATAACACCAGATAGGTACC Chr2 |
| [3]   | 100424006 AATCAATCTCCTCTGC...GAAGACCAGTGGAGATG Chr3 |
| [4]   | 92728752 CTAAACCCCAAGCCGAA...CATACCCCATGTGTTAT Chr4 |
| [5]   | 100743485 AATTTTGAGGAAATCGA...CAAGCATTGGCAACTC Chr5 |

... ...

Figure 30. Viewing Ms_genome.fasta in RStudio using Biostrings package.

> vcf_raw_data
***** Object of Class vcfR *****
163 samples
1 CHROMs
118,252 variants
Object size: 822.9 Mb
7.101 percent missing data
*****

Figure 31. Statistics of Chr1.vcf.filtered.500M.vcf shown by vcfR.

7.3 Faster Access through Command-line-based Database Frontend
8. Availability
The Alfalfa Database Project, including the Python, Java, and SQL scripts used during the development stage, the Flask web application, and the Java command line application are all open-sourced and posted on our GitHub repository:

https://github.com/ShichenQiao/Alfalfa_Database.git

However, up to the point we post this paper, we are not authorized to reveal the original dataset, our intermediate datafiles, and the resulting SQLite backend database to the public.

The service of our web application is only available at the Local Area Network (LAN) of China Agriculture University at 202.112.170.236:5000.

9. Authors’ Contributions
SQ designed the backend SQLite database and the frontend applications, developed the Python, Java, and SQL scripts used during the development stage (except vcf.java as shown in Section 10.2), and wrote the frontend web application and the command line application. CS managed and filtered datafiles, wrote vcf.java, provided server support, processed the complete VCF files, and deployed our web application to the LAN. All of the authors have read and approved the final form of this paper.

10. Appendix: Complete Scripts

10.1 create_tables.sql

```sql
1. CREATE TABLE "ID_to_uniqueid" ( 
2. "ID" TEXT NOT NULL UNIQUE, 
3. "start" INTEGER NOT NULL UNIQUE, 
4. "end" INTEGER NOT NULL UNIQUE, 
5. PRIMARY KEY("start","ID","end") 
6. );
7.
8. CREATE TABLE "alltpm" ( 
9. "GenelD" TEXT NOT NULL UNIQUE, 
10. "150_trueleaf_rep1" REAL NOT NULL, 
11. "150_xia_rep1" REAL NOT NULL, 
12. "150_ziye_rep1" REAL NOT NULL, 
13. "fennie_rep1" REAL NOT NULL, 
14. "flower_bud_rep1" REAL NOT NULL, 
15. "flower_rep1" REAL NOT NULL, 
16. "flower_rep2" REAL NOT NULL, 
17. "flower_rep3" REAL NOT NULL, 
18. "flower_rep4" REAL NOT NULL, 
19. "flower_rep5" REAL NOT NULL, 
20. "flower_rep6" REAL NOT NULL, 
21. "flower_rep7" REAL NOT NULL, 
22. "flower_rep8" REAL NOT NULL, 
23. "germ_seed_rep1" REAL NOT NULL, 
24. "guojia_rep1" REAL NOT NULL, 
25. "lateral_root_rep1" REAL NOT NULL,
```
"leaf_bud_rep1" REAL NOT NULL,
"leaf_bud_rep2" REAL NOT NULL,
"leaf_bud_rep3" REAL NOT NULL,
"main_root_rep1" REAL NOT NULL,
"mature_leaf_rep1" REAL NOT NULL,
"MS_trueleaf_rep1" REAL NOT NULL,
"MS_xia_rep1" REAL NOT NULL,
"MS_ziye_rep1" REAL NOT NULL,
"nod_21_rep1" REAL NOT NULL,
"mature_leaf_rep2" REAL NOT NULL,
"mature_leaf_rep3" REAL NOT NULL,
"young_leaf_rep1" REAL NOT NULL,
"young_leaf_rep2" REAL NOT NULL,
"young_leaf_rep3" REAL NOT NULL,
"seed_rep1" REAL NOT NULL,
"stem_branch_rep1" REAL NOT NULL,
"stem_rep1" REAL NOT NULL,
"yige_rep1" REAL NOT NULL,
"yige_rep2" REAL NOT NULL,
"yige_rep3" REAL NOT NULL,
"uniqer_leaf_rep1" REAL NOT NULL,
"uniqer_leaf_rep2" REAL NOT NULL,
"uniqer_leaf_rep3" REAL NOT NULL,
PRIMARY KEY("GeneID")
);
CREATE TABLE "kegg_info" (
"ID_full" TEXT NOT NULL UNIQUE,
"kegg" TEXT NOT NULL,
PRIMARY KEY("ID_full"),
CONSTRAINT "FK_kegg" FOREIGN KEY("ID_full") REFERENCES "pros_info"("ID_full")
);

CREATE TABLE "pathway_info" (
"ID_full" TEXT NOT NULL UNIQUE,
"pathway" TEXT NOT NULL,
PRIMARY KEY("ID_full"),
CONSTRAINT "FK_pathway" FOREIGN KEY("ID_full") REFERENCES "pros_info"("ID_full")
);

CREATE TABLE "pfam_info" (
"ID_full" TEXT NOT NULL UNIQUE,
"pfam" TEXT NOT NULL,
PRIMARY KEY("ID_full"),
CONSTRAINT "FK_pfam" FOREIGN KEY("ID_full") REFERENCES "pros_info"("ID_full")
);

CREATE TABLE "pros_info" (
"unique_id" INTEGER NOT NULL UNIQUE,
"ID_full" TEXT NOT NULL UNIQUE,
"pros" TEXT NOT NULL,
CONSTRAINT "FK" FOREIGN KEY("unique_id") REFERENCES "gff"("unique_id"),
PRIMARY KEY("unique_id","ID_full")
);

CREATE TABLE sqlite_sequence(name,seq);

CREATE TABLE "vcf" (
"vcf_ID" INTEGER NOT NULL UNIQUE,
"CHROM" TEXT NOT NULL,
"POS" INTEGER NOT NULL,
"REF" TEXT NOT NULL,
"ALT" TEXT NOT NULL,
"C1" TEXT NOT NULL,
"C10" TEXT NOT NULL,
"C100" TEXT NOT NULL,
"C101" TEXT NOT NULL,
"C102" TEXT NOT NULL,
"C103" TEXT NOT NULL,
"C104" TEXT NOT NULL,
"C105" TEXT NOT NULL,
"C106" TEXT NOT NULL,
"C107" TEXT NOT NULL,
"C108" TEXT NOT NULL,
"C109" TEXT NOT NULL,
"C11" TEXT NOT NULL,
"C110" TEXT NOT NULL,
"C111" TEXT NOT NULL,
"C112" TEXT NOT NULL,
"C113" TEXT NOT NULL,
"C114" TEXT NOT NULL,
"C115" TEXT NOT NULL,
"C116" TEXT NOT NULL,
"C117" TEXT NOT NULL,
"C118" TEXT NOT NULL,
"C119" TEXT NOT NULL,
"C12" TEXT NOT NULL,
|   |   |
|---|---|
| 148. | "C120" TEXT NOT NULL, |
| 149. | "C121" TEXT NOT NULL, |
| 150. | "C122" TEXT NOT NULL, |
| 151. | "C123" TEXT NOT NULL, |
| 152. | "C124" TEXT NOT NULL, |
| 153. | "C125" TEXT NOT NULL, |
| 154. | "C126" TEXT NOT NULL, |
| 155. | "C127" TEXT NOT NULL, |
| 156. | "C128" TEXT NOT NULL, |
| 157. | "C129" TEXT NOT NULL, |
| 158. | "C13" TEXT NOT NULL, |
| 159. | "C130" TEXT NOT NULL, |
| 160. | "C131" TEXT NOT NULL, |
| 161. | "C132" TEXT NOT NULL, |
| 162. | "C133" TEXT NOT NULL, |
| 163. | "C134" TEXT NOT NULL, |
| 164. | "C135" TEXT NOT NULL, |
| 165. | "C136" TEXT NOT NULL, |
| 166. | "C137" TEXT NOT NULL, |
| 167. | "C14" TEXT NOT NULL, |
| 168. | "C15" TEXT NOT NULL, |
| 169. | "C16" TEXT NOT NULL, |
| 170. | "C17" TEXT NOT NULL, |
| 171. | "C18" TEXT NOT NULL, |
| 172. | "C19" TEXT NOT NULL, |
| 173. | "C2" TEXT NOT NULL, |
| 174. | "C20" TEXT NOT NULL, |
| 175. | "C21" TEXT NOT NULL, |
| 176. | "C22" TEXT NOT NULL, |
| 177. | "C23" TEXT NOT NULL, |
| 178. | "C24" TEXT NOT NULL, |
| 179. | "C25" TEXT NOT NULL, |
| 180. | "C26" TEXT NOT NULL, |
| 181. | "C27" TEXT NOT NULL, |
| 182. | "C28" TEXT NOT NULL, |
| 183. | "C29" TEXT NOT NULL, |
| 184. | "C3" TEXT NOT NULL, |
| 185. | "C30" TEXT NOT NULL, |
| 186. | "C31" TEXT NOT NULL, |
| 187. | "C32" TEXT NOT NULL, |
| 188. | "C33" TEXT NOT NULL, |
| 189. | "C34" TEXT NOT NULL, |
| 190. | "C35" TEXT NOT NULL, |
| 191. | "C36" TEXT NOT NULL, |
| 192. | "C37" TEXT NOT NULL, |
| 193. | "C38" TEXT NOT NULL, |
| 194. | "C39" TEXT NOT NULL, |
| 195. | "C4" TEXT NOT NULL, |
| 196. | "C40" TEXT NOT NULL, |
| 197. | "C41" TEXT NOT NULL, |
| 198. | "C42" TEXT NOT NULL, |
| 199. | "C43" TEXT NOT NULL, |
| 200. | "C44" TEXT NOT NULL, |
| 201. | "C45" TEXT NOT NULL, |
| 202. | "C46" TEXT NOT NULL, |
| 203. | "C47" TEXT NOT NULL, |
| 204. | "C48" TEXT NOT NULL, |
| 205. | "C49" TEXT NOT NULL, |
| 206. | "C5" TEXT NOT NULL, |
| 207. | "C50" TEXT NOT NULL, |
| 208. | "C51" TEXT NOT NULL, |
10.2 vcf.java

```java
import java.io.BufferedReader;
import java.io.FileInputStream;
import java.io.IOException;
import java.io.InputStreamReader;
import java.util.ArrayList;

public class vcf {

    public static void main(String[] args) throws IOException {
        String vcf = args[0]; // Usage: java vcf <path_to_vcf>
        FileInputStream fis = new FileInputStream(vcf);
        BufferedReader br = new BufferedReader(new InputStreamReader(fis));
        String line = null;
        int cnt = 1; // vcf_ID
        while ((line = br.readLine()) != null) {
            // ignore file headers and column names
            if (line.length() == 0 || line.charAt(0) == '#')
                continue;
            String[] strArrays = line.split("\s+");
            StringBuilder fullLine = new StringBuilder();
            int i = 9;
            // Read variation info
            fullLine.append(strArrays[0]); // CHROM
            fullLine.append("\t");
            fullLine.append(strArrays[1]); // POS
            fullLine.append("\t");
            fullLine.append(strArrays[3]); // REF
            fullLine.append("\t");
            fullLine.append(strArrays[4]); // ALT
            fullLine.append("\t");
            // Split REF into lists
            ArrayList<String> ref = new ArrayList<String>();
            ref.add(strArrays[3]);
            String[] s2 = strArrays[4].split("\,");
            for (String s1 : s2) {
                ref.add(s1);
            }
        }
    }
}
```
39. }
40. 
41. // Replace */* notations to <seq>/<seq> notation
42. while (i < 172) {
43.   String[] sa = strArrays[i].split(";");
44.   ++i;
45.   String[] sc = sa[0].split("/");
46.   if (sa[0].equals("./.")) {
47.     fullLine.append("./.");
48.   } else {
49.     int t1 = Integer.parseInt(sc[0]);
50.     int t2 = Integer.parseInt(sc[1]);
51.     // Get correct REF sequences from the ArrayList
52.     fullLine.append(ref.get(t1) + "+" + ref.get(t2));
53.   }
54.   fullLine.append("\t");
55. }
56. 
57. // Generate sql INSERT statement for the current variation record
58. String sql =
59.   "INSERT INTO vcf(vcf_ID,CHROM,POS,REF,ALT,C1,C10,C100,C101,C102,C103,C104,C105,C106,"
60. + "C107,C108,C109,C11,C110,C111,C112,C113,C114,C115,C116,C117,C118,C119,C12,C120,"
61. + "C121,C122,C123,C124,C125,C126,C127,C128,C129,C13,C130,C131,C132,C133,C134,C135,"
62. + "C136,C137,C138,C139,C140,C141,C142,C143,C144,C145,C146,C147,"
63. + "C148,C149,C150,C151,C152,C153,C154,C155,C156,C157,C158,C159,C160,C161,C162,C163,"
64. + "C164,C165,C166,C167,C168,C169,C17,C170,C171,C172,C173,C174,C175,C176,C177,C178,"
65. + "C179,C180,C181,C182,C183,"
66. + "C184,C185,C186,C187,C188,C189,C190,C191,C192,C193,C194,C195,C196,C197,C198,C199,"
67. + "L1,L10,L11,L12,L13,L14,L15,L16,L17,L18,L19,L2,L20,L21,L22,L27,L3,L32,L33,L4,L5,L6,L7,L8,L9,"
68. + "MsREF,zhonglan) VALUES(" + cnt++ + ",";
69. String[] x = fullLine.toString().split("\t");
70. for (String element : x) {
71.   sql += "\t" + element + ",";
72. }
73. sql = sql.substring(0, sql.length() - 1);
74. sql += ");";
75. System.out.println(sql); // save to output .sql file
76. }
77. 
78. fis.close();
79. }
80. }

10.3 app.py

1. from flask import Flask, render_template, request, redirect
2. import sqlite3
3. 
4. app = Flask(__name__)
5. 
6. # given a DNA seq, return its conjugate seq
7. def get_conjugate(seq):
8.   rev = str(seq)[::-1]
9.   ans = 
10.   for i in range(0, len(rev)):
11.     if rev[i] == 'A':
12.       ans += 'T'
13.     elif rev[i] == 'T':
14.       ans += 'A'
15.     elif rev[i] == 'C':
ans += 'G'

elif rev[i] == 'G':
    ans += 'C'

return ans

# home page with input place holders
@app.route('/', methods=['POST', 'GET'])
def home():
    return render_template('home.html', show_place_holder=True)

# query mode A, ID search
@app.route('/ID/', methods=['POST', 'GET'])
def mode_A():
    if request.method == 'POST':
        id = request.form['content1']  # gene ID input from html
        if len(id) != 13 and len(id) != 16 and len(id) != 20:
            return render_template('error.html', err_msg="Input gene ID was invalid.")
        conn = sqlite3.connect("/static/sql/Alfalfa.db")
        cu = conn.cursor()
        # query pros_info
        sql = "SELECT * FROM pros_info WHERE ID_full LIKE " + id + "%%" + ""
        cu.execute(sql)
        pros_infos = cu.fetchall()
        if len(pros_infos) == 0:
            return render_template('error.html', err_msg="Gene ID not found")
        show_annotate = True
        if len(annotate_infos) == 0:
            show_annotate = False
        # query go_info, interpro_info, kegg_info, pathway_info, and pfam_info
        have_info = False
        pri_id = id
        if len(id) != 13:
            pri_id = id[0:13]
        sql = "SELECT pros_info.ID_full, pros_info.unique_id, kegg_info.kegg, pathway_info.pathway,"  
        sql += "INTERPRO_INFO.interpro, pfam_info pfam FROM pros_info" + 
        sql += "LEFT OUTER JOIN go_info ON pros_info.ID_full = go_info.ID_full" + 
        sql += "LEFT OUTER JOIN interpro_info ON pros_info.ID_full = interpro_info.ID_full" + 
        sql += "LEFT OUTER JOIN kegg_info ON pros_info.ID_full = kegg_info.ID_full" + 
        sql += "LEFT OUTER JOIN pathway_info ON pros_info.ID_full = pathway_info.ID_full" + 
        sql += "LEFT OUTER JOIN pfam_info ON pros_info.ID_full = pfam_info.ID_full" + 
        sql += "WHERE pros_info.ID_full LIKE " + pri_id + "%%" + ""
        cu.execute(sql)
        infos = cu.fetchall()
        # filter infos, only keep rows with at least one annotation that is not None
        info = []
        for x in infos:
            for i in range(2, 7):
if x[i] != None:
    info = x
    break

# process kegg
kegg_list = None
if len(info) > 0:
    if info[2] != None:
        kegg_list = str(info[2]).split(";")
if kegg_list != None:
    for i in range(0, len(kegg_list)):
        kegg_list[i] = kegg_list[i][6:]
    have_info = True

# process pathway
temp_list = None
pathway_list = []
if len(info) > 0:
    if info[3] != None:
        temp_list = str(info[3]).split(";")
if temp_list != None:
    for i in range(0, len(temp_list)):
        # exclude KEGG duplicates
        if not ("KEGG" in temp_list[i][0:4]):
            pathway_list.append(temp_list[i])
    if len(pathway_list) == 0:
        pathway_list = None
    else:
        have_info = True

# process go
go_list = None
if len(info) > 0:
    if info[4] != None:
        go_list = str(info[4]).split(";")
if go_list != None:
    for i in range(0, len(go_list)):
        go_list[i] = go_list[i][3:]
    have_info = True

# process interpro
interpro_list = None
if len(info) > 0:
    if info[5] != None:
        interpro_list = str(info[5]).split("|")
if interpro_list != None:
    have_info = True

# process pfam
pfam_list = None
if len(info) > 0:
    if info[6] != None:
        pfam_list = str(info[6]).split(";")
if pfam_list != None:
    have_info = True

# get 16-character gene ID
gene_ID = None
if have_info:
    gene_ID = str(info[0])[0:16]
41

```python
41. # query gff
42. sql = "SELECT * FROM gff AS G, ID_to_uniqueid AS I WHERE I.ID =" + pri_id + " AND G.unique_id >= I.start AND G.unique_id <= Lend;"
43. cu.execute(sql)
44. temp_list = cu.fetchall()
45. gffs = []
46. for x in temp_list:
47.     if id in x[9]:
48.         gffs.append(x)
49. # replace by DNA conjugates when needed
50. for i in range(0, len(gffs)):
51.     if gffs[i][7] == ":"
52.         temp = list(gffs[i])
53.         temp[10] = get_conjugate(temp[10])
54.         gffs[i] = tuple(temp)
55. # add combined CDS sequences to pros_info
56. cds = []
57. for x in gffs:
58.     if x[3] == "CDS":
59.         cds.append(x)
60.     cds_combined = ""
61. for x in cds:
62.         temp_list = list(pros_infos[i])
63.         if str(pros_infos[i][1]) in str(x[9]):
64.             cds_combined += x[10]
65.         temp_list.append(cds_combined)
66.         pros_infos[i] = tuple(temp_list)
67. # get transcriptome data from Table alltpm
68. sql = "SELECT * FROM alltpm WHERE GeneID = " + pri_id + ".01;"
69. cu.execute(sql)
70. temp_list = cu.fetchall()[0]
71. transcriptome_list = []
72. for i in range(1, len(temp_list)):
73.     transcriptome_list.append(temp_list[i])
74. # print to html page
75. if show_annotate:
76.     if have_info:
77.         return render_template('home.html', mode="A", id=id, show_annotate=show_annotate, annotate_infos=annotate_infos, temp=annotate_infos[0], annotate_span=len(annotate_infos)+1, gene_ID=gene_ID, have_info=have_info, kegg_list =kegg_list, pathway_list=pathway_list, go_list=go_list, interpro_list=interpro_list, pfam_list=pfam_list, pros_infos=pros_infos, gffs=gffs, len_gff=len(gffs), transcriptome_list=transcriptome_list)
78.     else:
79.         return render_template('home.html', mode="A", id=id, show_annotate=show_annotate, annotate_infos=annotate_infos, temp=annotate_infos[0], annotate_span=len(annotate_infos)+1, have_info=have_info, pros_infos=pros_infos, gffs=gffs, len_gff=len(gffs), transcriptome_list=transcriptome_list)
80. else:
81.     return render_template('home.html', mode="A", id=id, show_annotate=show_annotate, gene_ID=gene_ID, have_info=have_info, kegg_list=kegg_list, pathway_list=pathway_list, go_list=go_list, interpro_list=interpro_list, pfam_list=pfam_list, pros_infos=pros_infos, gffs=gffs, len_gff=len(gffs), transcriptome_list=transcriptome_list)
82. else:
83.     return render_template('home.html', mode="A", id=id, show_annotate=show_annotate, gene_ID=gene_ID, have_info=have_info, pros_infos=pros_infos, gffs=gffs, len_gff=len(gffs), transcriptome_list=transcriptome_list)
84. return redirect('/')
85. # query mode B, keywords search
```
@app.route('/annotation/', methods=['POST', 'GET'])
def mode_B():
    if request.method == 'POST':
        keywords = request.form['content2']  # keywords input from html

        if len(keywords) < 1:
            return render_template('error.html', err_msg="No keywords entered.")

        # query annotate_info
        conn = sqlite3.connect("./static/sql/Alfalfa.db")
        cu = conn.cursor()
        sql = "SELECT * FROM annotate_info WHERE info1 LIKE " + keywords + " OR info2 LIKE " + keywords + "%;"
        cu.execute(sql)
        ans_list = cu.fetchall()

        if len(ans_list) == 0:
            return render_template('error.html', err_msg="No such keywords found in annotation records.")

        # combine rows with the same gene ID by calculating rowspan value for html
        rowspan = []
        temp = ans_list[0][0]
        cnt = 0
        l = 0
        for ans in ans_list:
            if ans[0] == temp:
                cnt += 1
            else:
                if len(rowspan) == 0:
                    rowspan.append(cnt)
                    for i in range(1, cnt):
                        rowspan.append(0)
                else:
                    l += 1
                    rowspan.append(cnt + 1)
                    for i in range(1, cnt + 1):
                        rowspan.append(0)
                    temp = ans[0]
                    cnt = 0
                    rowspan.append(len(ans_list) - len(rowspan))
        for i in range(0, len(ans_list) - len(rowspan)):
            rowspan.append(0)

        # add rowspan information to each row of query results
        for i in range(0, len(ans_list)):
            temp_list = []
            temp_list.append(rowspan[i])
            ans_list[i] += tuple(temp_list)

        # print to html page
        temp_list = []
        temp_list.append(rowspan[i])
        ans_list[i] += tuple(temp_list)

        # query mode C, VCF search
        @app.route('/VCF/', methods=['POST', 'GET'])
def mode_C():
    if request.method == 'POST':
        seqid = request.form['content3']  # seqid input from html
302. low = request.form['content4']  # low input from html
303. high = request.form['content5']  # high input from html
304.
305. if len(seqid) != 4 or len(low) < 1 or len(high) < 1 or int(low) < 1 or int(high) < 1 or int(low) >= int(high):
306.     return render_template('error.html', err_msg="Invalid VCF query.")
307.
308. # default range is set to be 1 ~ 3000 bp
309. if int(high) - int(low) >= 3000:
310.     return render_template('error.html', err_msg="Please select a range that is less than or equals to 3000bp.")
311.
312. conn = sqlite3.connect("/static/sql/Alfalfa.db")
313. cu = conn.cursor()
314.
315. # setup ranges from sql results
316. max_pos = 148124263  # get_max_vcfID.sql
317. arr = [1, 17551074, 32991065, 52346699, 69388115, 89626087, 113452234, 132015073, max_pos + 1]  # get_min_vcfIDs.sql
318.
319. # get a smaller vcf_ID range to query from
320. temp_low = str(arr[int(seqid[3:4]) - 1])
321. temp_high = str(arr[int(seqid[3:4])])
322.
323. sql = "SELECT vcf_ID FROM vcf_jump_table WHERE vcf_ID BETWEEN " + temp_low + " AND " + temp_high + " AND POS <= " + low + " ORDER BY vcf_ID DESC LIMIT 1;"
324. cu.execute(sql)
325. temp = cu.fetchall()
326. start = str(temp_low)
327. if len(temp) != 0:
328.     start = str(list(temp[0][0])
329. end = str(temp_high)
330. if len(temp) != 0:
331.     end = str(list(temp[0][0])
332.
333. # query VCF records
334. sql = "SELECT * FROM vcf WHERE vcf_ID BETWEEN " + start + " AND " + end + " AND POS BETWEEN " + low + " AND " + high + ";"
335. cu.execute(sql)
336. vcfs = cu.fetchall()
337.
338. if len(vcfs) == 0:
339.     return render_template('error.html', err_msg="No VCF entry found.")
340.
341. # get DNA sequence between low and high
342. file_name = "./static/DNA/" + seqid + ".txt"
343. f = open(file_name, 'r')
344. data = f.read()
345. f.close()
346. seq = data[int(low) - 1:int(high)]
347.
348. # print to html page
349. return render_template('home.html', mode="C", seqid=int(seqid[3:]), low=low, high=high, length=len(seq), DNA=seq, vcfs=vcfs, l=len(vcfs))
350. return redirect("")
351.
352. if __name__ == "__main__":
353.     app.run(host='0.0.0.0', port='5000', debug = True)
10.4  base.html

1. `<!DOCTYPE html>`
2. `<html lang="en">`
3. `<head>`
4. `<meta charset="UTF-8">`
5. `<meta name="viewport" content="width=device-width, initial-scale=1.0">`
6. `<meta http-equiv="X-UA-Compatible" content="ie=edge">`
7. `<link rel="stylesheet" href="{{ url_for('static', filename='css/main.css') }}">`
8. `{% block head %}`
9. `</head>`
10. `<body>`
11. `{% block body %}`
12. `</body>`
13. `</html>`

10.5  home.html

1. `{% extends 'base.html' %}`
2. `{% block head %}`
3. `Alfalfa Database<title>`
4. `<script src="https://cdn.staticfile.org/echarts/4.3.0/echarts.min.js"></script>`
5. `{% endblock %}`
6. `{% block body %}`
7. `<div><h1>Alfalfa Database</h1></div>`
8. `{% if show_place_holder %}`
9. `<div>`
10. `<form action="/ID/" method='POST'>`
11. `Gene ID:`
12. `<input type="text" name="content1" id="content1" placeholder="MsG1A10000001">`
13. `<input type="submit" value="Search gene ID">`
14. `</form>`
15. `</p></div>`
16. `<form action="/annotation/" method='POST'>`
17. `Annotation Keywords:`
18. `<input type="text" name="content2" id="content2" placeholder="Heat shock factor">`
19. `<input type="submit" value="Search annotation">`
20. `</form>`
21. `</p></form>`
22. `<form action="/VCF/" method='POST'>`
23. `Seqid:`
24. `<select name="content3" id="content3">`
25. `<option value="Chr1">Chr1</option>`
26. `<option value="Chr2">Chr2</option>`
27. `<option value="Chr3">Chr3</option>`
28. `<option value="Chr4">Chr4</option>`
29. `<option value="Chr5">Chr5</option>`
30. `<option value="Chr6">Chr6</option>`
31. `<option value="Chr7">Chr7</option>`
32. `<option value="Chr8">Chr8</option>`
33. `</select>`
34. `POS from:`
35. `<input type="text" name="content4" id="content4" placeholder="1">`
36. `to`
37. `<input type="text" name="content5" id="content5" placeholder="300">`
38. `<input type="submit" value="Search VCF">`
(Range <= 3000bp)

</form>

<p>
</p>

<form action="/" method='POST'>

<input type="submit" value="/Reset"/>
</form>

</div>

<p>

</div>

<!-- Users' Guide:-->

<h2>1. An valid Gene ID could have 13 (MsG1A10000001), 16 (MsG1A10000001.01), or 20 (MsG1A10000001.01.
T01) characters.</h2>

<h3>2. When querying about a gene ID, all related genome sequencing, transcriptome sequencing, and annotation dat
a would be shown.</h3>

<h3>3. When querying about a keyword, the gene IDs of all annotation records containing that keyword would be fou
nd (case insensitive).</h3>

<h3>4. To improve page loading time, the range of VCF queries was restricted to be 1 ~ 3000 basepairs.</h3>

</div>

<p></p>

<form action="/ID/" method='POST'>

Gene ID:
</form>

<input type="text" name="content1" id="content1" value={{ id }}/>

<input type="submit" value="Search gene ID"/>
</form>

<p>

</p>

<form action="/annotation/" method='POST'>

Annotation Keywords:
</form>

<input type="text" name="content2" id="content2" value="/"+
</form>

<p>

</p>

<form action="/VCF/" method='POST'>

Seqid:
</form>

<select name="content3" id="content3">
</select>

<script type="text/javascript">

var num = '{ seqid }';

for(var i = 1; i < num; i++) {

if(num.toString() == num)

document.write("<option value="Chr" + i + ">
</option> ");

else

document.write("<option value="Chr" + i + ">Chr" + i + 
</option> ");

}
</script>

<p>

</p>

<form action="/pos/" method='POST'>

POS from:
</form>

<input type="text" name="content4" id="content4" value={{ low }}/>

<input type="text" name="content5" id="content5" value={{ high }}/>

</form>
<input type="submit" value="Search VCF">

<form action="/" method='POST'>
  <p></p>
  <input type="submit" value="Reset">
</form>

{% if mode == "A" %}

<div class="container">
  <h2>Query Results:</h2>
</div>

{% if have_info %}
<table>
<thead><th>Annotation Links</th></thead>
<tbody><tr><td>
  <div class="container" style="text-align: left;">
    <h3>Gene ID: {{ gene_ID }}</h3>
    {% if kegg_list != None %}
    <h3>KEGG: </h3>
    {% for kegg in kegg_list %}
    <a href="https://www.kegg.jp/entry/K{{ kegg }}" target="_blank">{{ kegg }}</a>;  
    {% endfor %}
    </h3>
    {% endif %}
  </div>
</td></tr>
</tbody><tr><td>
  <div class="container" style="text-align: left;">
    <h3>Pathway: </h3>
    {% for pathway in pathway_list %}
    <a style="text-decoration: underline;">{{ pathway }}</a>
    {% endfor %}
  </div>
</td></tr>
</tbody></table>

{% endif %}

{% endif %}

{% if go_list != None %}
<h3>GO: </h3>
{% for go in go_list %}
<a href="http://amigo.geneontology.org/amigo/term/GO:{{ go }}" target="_blank">{{ go }}</a>
{% endfor %}
</h3>
{% endif %}

{% endif %}

{% if interpro_list != None %}
<h3>Interpro: </h3>
<script type="text/javascript">
  var jsVar = '{% for interpro in interpro_list %}'
  var i = jsVar.length;
  var description = jsVar.substr(0, i - 11)
  var code = jsVar.substr(i - 10, 9)
  document.write(description + "(<a href="https://www.ebi.ac.uk/interpro/entry/InterPro" target="_blank">" + code + "</a>);")
</script>
{% endif %}

{% endif %}
{% if pfam_list != None %}
<h3>Pfam:</h3>
{% for pfam in pfam_list %}
<a href="https://pfam.xfam.org/family/{{ pfam }}" target="_blank">{{ pfam }}</a>;
{% endfor %}
{% endif %}
</div>
</td></tr></tbody>
</table>
<p></p>
{% endif %}

<!-- each gene ID query should have at most one set of annotations -->
{% if show_annotate %}
<table>
<thead>
<th>Gene ID</th>
<th>db Name</th>
<th>Info 1</th>
<th>Info 2</th>
</thead>
<tbody>
{% if annotate_span != 0 %}
<td rowspan="{{ annotate_span }}">{{ temp[0] }}</td>
{% endif %}
{% for annotate_info in annotate_infos %}
<tr>
<td>{{ annotate_info[1] }}</td>
<td>{{ annotate_info[2] }}</td>
<td>{{ annotate_info[3] }}</td>
</tr>
{% endfor %}
</tbody>
</table>
<p></p>
{% endif %}

<!-- visualize transcriptome data using echarts -->
<div id="main" style="border: 2px solid #aaa; width: 98%; height:500px; background-color: rgb(212, 239, 248); overflow-x: scroll;"></div>

<script type="text/javascript">
// convert Python list to js array
var jsVar = '{{ transcriptome_list }}'.toString();
var jsArray = new Array();
jsVar = jsVar.substr(1, jsVar.length - 2);
while(jsVar.length > 0) {
    var index = jsVar.indexOf(',');
    if(index == -1) break;
    var temp = jsVar.substr(0, index - 1);
    jsArray.push(Number(temp));
    jsVar = jsVar.substr(index + 2);
}
jsArray.push(Number(jsVar));

// render the echarts
var myChart = echarts.init(document.getElementById('main'));
var option = {
    title: {
        ...
Expression in different Tissues

```javascript
},
tooltip: {},
xAxis: {
  axisLabel: {
    show: true,
    interval: 0,
    rotate: 45
  },
  data: ['150_trueleaf_rep1','150_xia_rep1','150_ziye_rep1','fennie_rep1','flower_bud_rep1','flower_rep1',
  'flower_rep2','flower_rep3','flower_rep4','flower_rep5','flower_rep6','flower_rep7','flower_rep8','germ_seed_rep1',
  'guojia_rep1','lateral_root_rep1','leaf_bud_rep1','leaf_bud_rep2','leaf_bud_rep3', 'main_root_rep1','mature_leaf_rep1','mature_leaf_rep2','mature_leaf_rep3',
  'MS_trueleaf_rep1','MS_xia_rep1','MS_ziye_rep1','nod_21_rep1','nod_28_rep1',
  'seed_rep1','stem_branch_rep1','stem_rep1','yige_rep1','yige_rep2','yige_rep3', 'young_leaf_rep1','young_leaf_rep2',
  'young_leaf_rep3']
},
yAxis: {},
grid: {
  y2: 80
},
series: [{
  name: 'TPM',
  type: 'bar',
  data: jsArray
}],
myChart.setOption(option);
</script>
<p></p>

<!-- each valid 20-character gene ID should have one set of pros_info with complete CDS seq -->
<table>
<thead><th>Gene ID: {{ pros_info[1] }}</th></thead>
<tbody>
<tr style='background-color: bisque;'>
<td>Protein Sequence</td>
<td>{{ pros_info[2] }}</td>
<td>Complete CDS Sequence</td>
</tr>
</tbody>
</table>

<!-- each valid 20-character gene ID should have some gff records -->
<div><h3>{{ len_gff }}</h3> gff entries were found:</div>
<table>
<thead><th>Seqid</th><th>Source</th><th>Type</th><th>Start</th><th>End</th><th>Score</th><th>Strand</th><th>Phase</th><th>Attributes</th>
</thead>
<tbody>
<tr style='background-color: bisque;'>
<td>{{ gff[1] }}</td>
</tr>
</tbody>
</table>
```
<table>
  <thead>
    <tr>
      <th>Gene ID</th>
      <th>db Name</th>
      <th>Info 1</th>
      <th>Info 2</th>
    </tr>
  </thead>
  <tbody>
    {% for ans in ans_list %}
      <tr>
        <td> Gene ID</td>
        <td> db Name</td>
        <td> Info 1</td>
        <td> Info 2</td>
      </tr>
    {% endfor %}
  </tbody>
</table>

---

<h3>Note: Clicking on any of the sequences could select the entire sequence for copying purposes.</h3>

---

<h3>Note: Double-clicking on any of the sequences could select the entire sequence for copying purposes.</h3>

---

<h3>Note: Double-clicking on any of the sequences could select the entire sequence for copying purposes.</h3>

---

<h3>Note: Clicking on the question mark next to a gene ID would redirect to the query results about that gene ID.</h3>

---

<h3>Note: Double-clicking on any of the sequences could select the entire sequence for copying purposes.</h3>
| Seqid | Start | End | Total Length |
|-------|-------|-----|--------------|
| Chr{{seqid}} | low | high | length bp |

---

### Main DNA Sequence

| POS | REF | ALT | C1 | C10 | C100 | C101 | C102 | C103 | C104 | C105 | C106 | C107 | C108 | C109 | C11 | C110 | C111 | C112 | C113 | C114 | C115 | C116 | C117 | C118 | C119 | C12 | C120 | C121 | C122 | C123 | C124 |
| No. | Column Name |
|-----|-------------|
| 395 | C125        |
| 396 | C126        |
| 397 | C127        |
| 398 | C128        |
| 399 | C129        |
| 400 | C13         |
| 401 | C130        |
| 402 | C131        |
| 403 | C132        |
| 404 | C133        |
| 405 | C134        |
| 406 | C135        |
| 407 | C136        |
| 408 | C137        |
| 409 | C138        |
| 410 | C139        |
| 411 | C140        |
| 412 | C141        |
| 413 | C142        |
| 414 | C143        |
| 415 | C144        |
| 416 | C145        |
| 417 | C146        |
| 418 | C147        |
| 419 | C148        |
| 420 | C149        |
| 421 | C150        |
| 422 | C151        |
| 423 | C152        |
| 424 | C153        |
| 425 | C154        |
| 426 | C155        |
| 427 | C156        |
| 428 | C157        |
| 429 | C158        |
| 430 | C159        |
| 431 | C160        |
| 432 | C161        |
| 433 | C162        |
| 434 | C163        |
| 435 | C164        |
| 436 | C165        |
| 437 | C166        |
| 438 | C167        |
| 439 | C168        |
| 440 | C169        |
| 441 | C170        |
| 442 | C171        |
| 443 | C172        |
| 444 | C173        |
| 445 | C174        |
| 446 | C175        |
| 447 | C176        |
| 448 | C177        |
| 449 | C178        |
| 450 | C179        |
| 451 | C180        |
| 452 | C181        |
| 453 | C182        |
| 454 | C183        |
| 455 | C184        |
|   |   |   |   |   |
|---|---|---|---|---|
| 456. |  <th>C57</th> |   |   |   |
| 457. |  <th>C58</th> |   |   |   |
| 458. |  <th>C59</th> |   |   |   |
| 459. |  <th>C6</th> |   |   |   |
| 460. |  <th>C60</th> |   |   |   |
| 461. |  <th>C61</th> |   |   |   |
| 462. |  <th>C62</th> |   |   |   |
| 463. |  <th>C63</th> |   |   |   |
| 464. |  <th>C64</th> |   |   |   |
| 465. |  <th>C65</th> |   |   |   |
| 466. |  <th>C66</th> |   |   |   |
| 467. |  <th>C67</th> |   |   |   |
| 468. |  <th>C68</th> |   |   |   |
| 469. |  <th>C69</th> |   |   |   |
| 470. |  <th>C7</th> |   |   |   |
| 471. |  <th>C70</th> |   |   |   |
| 472. |  <th>C71</th> |   |   |   |
| 473. |  <th>C72</th> |   |   |   |
| 474. |  <th>C73</th> |   |   |   |
| 475. |  <th>C74</th> |   |   |   |
| 476. |  <th>C75</th> |   |   |   |
| 477. |  <th>C76</th> |   |   |   |
| 478. |  <th>C77</th> |   |   |   |
| 479. |  <th>C78</th> |   |   |   |
| 480. |  <th>C79</th> |   |   |   |
| 481. |  <th>C8</th> |   |   |   |
| 482. |  <th>C80</th> |   |   |   |
| 483. |  <th>C81</th> |   |   |   |
| 484. |  <th>C82</th> |   |   |   |
| 485. |  <th>C83</th> |   |   |   |
| 486. |  <th>C84</th> |   |   |   |
| 487. |  <th>C85</th> |   |   |   |
| 488. |  <th>C86</th> |   |   |   |
| 489. |  <th>C87</th> |   |   |   |
| 490. |  <th>C88</th> |   |   |   |
| 491. |  <th>C89</th> |   |   |   |
| 492. |  <th>C9</th> |   |   |   |
| 493. |  <th>C90</th> |   |   |   |
| 494. |  <th>C91</th> |   |   |   |
| 495. |  <th>C92</th> |   |   |   |
| 496. |  <th>C93</th> |   |   |   |
| 497. |  <th>C94</th> |   |   |   |
| 498. |  <th>C95</th> |   |   |   |
| 499. |  <th>C96</th> |   |   |   |
| 500. |  <th>C97</th> |   |   |   |
| 501. |  <th>C98</th> |   |   |   |
| 502. |  <th>C99</th> |   |   |   |
| 503. |  <th>L1</th> |   |   |   |
| 504. |  <th>L10</th> |   |   |   |
| 505. |  <th>L11</th> |   |   |   |
| 506. |  <th>L12</th> |   |   |   |
| 507. |  <th>L13</th> |   |   |   |
| 508. |  <th>L15</th> |   |   |   |
| 509. |  <th>L16</th> |   |   |   |
| 510. |  <th>L17</th> |   |   |   |
| 511. |  <th>L18</th> |   |   |   |
| 512. |  <th>L19</th> |   |   |   |
| 513. |  <th>L2</th> |   |   |   |
| 514. |  <th>L20</th> |   |   |   |
| 515. |  <th>L21</th> |   |   |   |
| 516. |  <th>L22</th> |   |   |   |
<thead>
<tr>
<th>L27</th>
<th>L3</th>
<th>L32</th>
<th>L33</th>
<th>L4</th>
<th>L5</th>
<th>L6</th>
<th>L7</th>
<th>L8</th>
<th>MsREF</th>
<th>zhonglan</th>
</tr>
</thead>
<tbody>
{% for vcf in vcfss %}
<tr>
<td style="background-color: bisque;">{{ vcf[2] }}</td>
<td style="background-color: rgb(166, 238, 95); overflow-x: scroll;">{{ vcf[3] }}</td>
<td style="background-color: rgb(245, 104, 238); overflow-x: scroll;">{{ vcf[4] }}</td>
{% for x in vcf[5:] %}
<!-- color each table cell accordingly -->
<script type="text/javascript">
var jsVar = '{{ x }}';
if(jsVar === "/") {
    document.write("<td style="background-color: rgb(166, 238, 95); overflow-x: scroll;">" + jsVar + "</td>");
} else {
    var index = jsVar.indexOf("/");
    var REF = '{{ vcf[3] }}';
    if(jsVar.substr(0, index) === REF) {
        if(jsVar.substr(index + 1) === REF) {
            document.write("<td style="background-color: rgb(166, 238, 95); overflow-x: scroll;">" + jsVar + "</td>");
        } else {
            document.write("<td style="background-color: rgb(255, 172, 17); overflow-x: scroll;">" + jsVar + "</td>");
        }
    } else {
        document.write("<td style="background-color: yellow; overflow-x: scroll;">" + jsVar + "</td>");
    }
}
{% endfor %}
</script>
{% endfor %}
</tr>
</tbody>
</table>
<p></p>
<div><h3>
Legend:
</h3>
<span style="background-color: rgb(166, 238, 95);">REF gene type</span>
Het with a REF; Het without a REF; Homo mutation

Note: Clicking on any part of the table above could select the entire table so that it could be copied to an Excel spreadsheet.

10.6 error.html

{% extends 'base.html' %}
{% block head %}
<title>Alfalfa Database</title>
{% endblock %}
{% block body %}
<div><h1>Alfalfa Database</h1></div>
<form action="/ID/" method = 'POST'>
Gene ID:<input type="text" name="content1" id="content1" placeholder="MsG1A10000001" />
<input type="submit" value="Search gene ID" />
</form>
<form action="/annotation/" method = 'POST'>
Annotation Keywords:<input type="text" name="content2" id="content2" placeholder="Heat shock factor" />
<input type="submit" value="Search annotation" />
</form>
<form action="/VCF/" method = 'POST'>
Seqid:
<select name="content3" id="content3">
<option value="Chr1">Chr1</option>
<option value="Chr2">Chr2</option>
<option value="Chr3">Chr3</option>
<option value="Chr4">Chr4</option>
<option value="Chr5">Chr5</option>
<option value="Chr6">Chr6</option>
<option value="Chr7">Chr7</option>
<option value="Chr8">Chr8</option>
</select>
POS from:<input type="text" name="content4" id="content4" placeholder="1" />
to<input type="text" name="content5" id="content5" placeholder="300" />
<input type="submit" value="Search VCF" />
(Range <= 3000bp)
</form>
<p></p>
<form action="/" method = 'POST'>
<input type="submit" value="Reset" />
</form>
<p></p>
</form>
10.7 main.css

```css
body, html {
  font-family: Arial, Helvetica, sans-serif;
  background-color: rgb(228, 226, 226);
}

table, td, th {
  border: 2px solid #aaa;
}
table {
  margin: auto;
  border-collapse: collapse;
  background-color: rgb(212, 239, 248);
  width: 98%;
}
thead tr th {
  background: #B8C1C8;
  position: sticky;
  top: -2px;
}
tbody {
  overflow-y: scroll;
}
tr {
  height: 30px;
}
  td {
    text-align: center;
    max-width: 100px;
  }
  .seq {
    word-wrap: break-word;
    word-break: break-all;
    text-align: left;
  }
  div {
    margin: auto;
    text-align: center;
  }
  .container {
    width: 80%;
    max-height: 800px;
    overflow-y: scroll;
  }
  tr:hover {
    background-color: rgb(228, 250, 234);
  }
```
10.8 Alfalfa_Database.java

```java
import java.io.File;
import java.sql.*;
import org.sqlite.JDBC;
import java.util.Arrays;
import java.util.HashSet;
import java.util.Scanner;

public class database {
    // given a DNA seq, return its conjugate seq
    private static String get_conjugate(String seq) {
        String ans = "";
        for (int i = seq.length() - 1; i >= 0; i--) {
            if (seq.charAt(i) == 'A') ans += 'T';
            else if (seq.charAt(i) == 'T') ans += 'A';
            else if (seq.charAt(i) == 'C') ans += 'G';
            else if (seq.charAt(i) == 'G') ans += 'C';
        }
        return ans;
    }

    public static void main(String[] args) {
        try {
            // database setup
            Scanner sc = new Scanner(System.in);
            String dbPath = "./Alfalfa.db";
            File dbFile = new File(dbPath);
            while (!dbFile.exists()) {
                System.out.println("*** Alfalfa.db is not found under the current path: ***");
                System.out.println("** " + dbPath + " **");
                System.out.println("*** Please enter a correct path to Alfalfa.db: ***");
                System.out.print(">");
                dbPath = sc.nextLine();
                while (!dbPath.contains("/Alfalfa.db")) {
                    System.out.println("*** The path, absolute or relative, should ends by="/Alfalfa.db" ***");
                    System.out.println("> ");
                    dbPath = sc.nextLine();
                }
                dbFile = new File(dbPath);
            }
            String sql = "jdbc:sqlite:" + dbPath;
            Class.forName("org.sqlite.JDBC");
            Connection conn = DriverManager.getConnection(sql);
            Statement stat = conn.createStatement();
            ResultSet rs;
            int cnt = 0;
            while (true) {
                // repeat these prompts after each query until option C is selected
                System.out.println("***");
            }
        }
    }
}
```
if
//
String
System.out.println("[A] Query about all information in Alfalfa.db related to a gene ID");
System.out.println("[B] Query about IDs of all genes whose annotations include a keywords");
System.out.println("[C] Query about all variations between two points on a chromosome");
System.out.println("[D] Exit");
System.out.println(">");
String option = sc.nextLine();

// exit app
if (option.substring(0, 1).toUpperCase().equals("D")) {
  System.out.println("Thank you for using the Alfalfa database.");
  System.out.println("******************************************************************************");
  break;
}

// query mode A
if (option.substring(0, 1).toUpperCase().equals("A")) {
  System.out.println("Please enter a ID");
  System.out.println(">");
  String query_id = sc.nextLine();

  // check validity of query_id, process it if needed
  String extension; // "." if no extension
  extension = ";
  if (query_id.length() == 16) {
    query_id = query_id.substring(0, 13); // all records in gff have .01
  } else if (query_id.length() == 20) {
    // separate main ID and its extension
    extension = ";
    query_id = query_id.substring(13);
  } else if (query_id.length() != 13) {
    System.out.println("** The ID should have 13, 16, or 20 characters **");
    continue;
  }

  // query annotate_info
  rs = stat.executeQuery("SELECT * FROM annotate_info WHERE ID_full LIKE \"%\" + query_id + extension + "%\"");
  cnt = 0;
  while (rs.next()) {
    if (cnt == 0)
      System.out.println("**** Results from annotate_info: ****");
    cnt++;
    System.out.println("id_full = " + rs.getString("id_full") + ",");
    System.out.println("db_name = " + rs.getString("db_name") + ",");
    System.out.println("info1 = " + rs.getString("info1") + ",");
    System.out.println("info2 = " + rs.getString("info2") + ");
  }
  System.out.println("**** " + cnt + ", related annotate_info were returned above ****\n");

  // query go_info, interpro_info, kegg_info, pathway_info, and pfam_info
  rs = stat.executeQuery("SELECT pros_info.ID_full, pros_info.unique_id, kegg_info.kegg, pathway_info.pathway," + " go_info.go, interpro_info.interpro, pfam_info.pfam FROM pros_info\n" + " LEFT OUTER JOIN go_info ON pros_info.ID_full = go_info.ID_full\n" + " LEFT OUTER JOIN interpro_info ON pros_info.ID_full = interpro_info.ID_full\n" + " LEFT OUTER JOIN kegg_info ON pros_info.ID_full = kegg_info.ID_full\n" + " LEFT OUTER JOIN pathway_info ON pros_info.ID_full = pathway_info.ID_full\n"
LEFT OUTER JOIN pfam_info ON pros_info.ID_full = pfam_info.ID_full
WHERE pros_info.ID_full LIKE "\m* + query_id + extension + "%\n":";

while (rs.next()) {
    if (rs.getString("source") != null || rs.getString("source") != null
        || rs.getString("kegg") != null || rs.getString("pathway") != null
        || rs.getString("pfam") != null) {
        cnt = 0;
        System.out.println("*** Annotations from other databases: ***");
        while (rs.next()) {
            System.out.println("< id_full = " + rs.getString("id_full") + ">");
            System.out.println("go = " + rs.getString("go"));
            System.out.println("interpro = " + rs.getString("interpro")");
            System.out.println("kegg = " + rs.getString("kegg"));
            System.out.println("pathway = " + rs.getString("pathway")");
            System.out.println("pfam = " + rs.getString("pfam")");
        }
        if (cnt == 0)
            System.out.println("*** Related annotations from other databases were returned above ***\n");
        // query pros_info
        rs = stat.executeQuery("SELECT * FROM pros_info WHERE ID_full LIKE "\m* + query_id + extension + "%\n":";
        cnt = 0;
        while (rs.next()) {
            System.out.println("*** Results from pros_info database: ****");
            cnt++;
            System.out.println("id_full = " + rs.getString("id_full") + ");
            System.out.println("pros = " + rs.getString("pros")");
        }
        if (cnt > 0)
            System.out.println("*** + cnt + " related pros_info were returned above ***\n");
        // query gff
        rs = stat.executeQuery("SELECT * FROM gff AS G, ID_to_uniqueid AS I WHERE I.ID ="\m* + query_id + " AND G.unique_id >= I.start AND G.unique_id <= I.end;"");
        cnt = 0;
        while (rs.next()) {
            if (!extension.equals("")) {
                String full_id = query_id + extension;
                String rst_id = rs.getString("attributes");
                if (!rst_id.contains(full_id))
                    continue;
            }
            cnt++;
            System.out.println("seqid = " + rs.getString("seqid") + ");
            System.out.println("source = " + rs.getString("source") + ");
            System.out.println("type = " + rs.getString("type") + ");
            System.out.println("start = " + rs.getString("start") + ");
            System.out.println("end = " + rs.getString("end") + ");
            System.out.println("score = " + rs.getString("score") + ");
            System.out.println("strand = " + rs.getString("strand") + ");
            System.out.println("phase = " + rs.getString("phase") + ");
            System.out.println("attributes = " + rs.getString("attributes") + ");
            String dna = rs.getString("DNA");
            // replace by DNA conjugates when needed
if (rs.getString("strand").equals("-"))
    dna = get_conjugate(dna);
System.out.println("DNA = " + dna);
if (cnt > 0)
    System.out.println("**" + cnt + " related gff records and their DNA sequences were returned above **");
else
    System.out.println("** NO related information was found in the Alfalfa database **");
rs.close();

if (option.substring(0, 1).toUpperCase().equals("B")
    System.out.println("Please enter some keywords:";
    System.out.print(">");
    String keywords = sc.nextLine();
    rs = stat.executeQuery("SELECT * FROM annotate_info WHERE info1 LIKE "%" + keywords + "%" OR info2 LIKE "%" + keywords + "%";");
cnt = 0;
    int number_of_unique_id = 0;
    HashSet<String> my_set = null;
while (rs.next()) {
    if (cnt == 0) {
        my_set = new HashSet<String>();
        number_of_unique_id = 0;
    }
    cnt++;
    if (my_set.add(rs.getString("id_full")))
        number_of_unique_id++;
    System.out.println("id_full = " + rs.getString("id_full") + ", ");
    System.out.println("db_name = " + rs.getString("db_name") + ", ");
    System.out.println("info1 = " + rs.getString("info1") + ", ");
    System.out.println("info2 = " + rs.getString("info2"));
}
if (cnt > 0)
    System.out.println("**" + cnt + " related annotate_info were returned above **");
    System.out.println("**" + number_of_unique_id + " unique gene IDs involved: **");
if (my_set != null && !my_set.isEmpty())
    System.out.println(my_set.toString());
else
    System.out.println("** NO related information was found in the Alfalfa database **");
rs.close();

// query mode C
if (option.substring(0, 1).toUpperCase().equals("C")
    System.out.println("Please enter a chromosome number (an integer between 1 and 8):";
    System.out.print(">");
    int num;
    try {
        num = Integer.parseInt(sc.nextLine());
        if (num < 1 || num > 8)
            System.out.println("** The chromosome number entered was invalid **");
        continue;
    } catch (NumberFormatException e) {
        System.out.println("** The chromosome number entered was invalid **");
        continue;
    }
// values from get_max_vcfID.sql and get_min_vcfIDs.sql
int max_pos = 148124263;
int[] arr = {1, 17551074, 32991065, 52346699, 69388115, 89626087, 113452234, 132015073};

int max_pos + 1

int vcf_ID_lower_bound = arr[num - 1];
int vcf_ID_upper_bound = arr[num] - 1;
int POS_lower_bound;
int POS_upper_bound;

try {
    System.out.println("Please enter a lower bound of POS (a positive integer):");
    System.out.print(">");
    POS_lower_bound = Integer.parseInt(sc.nextLine());
    if (POS_lower_bound < 1) {
        System.out.println("*** The lower bound entered was invalid ***");
        continue;
    }
    System.out.println("Please enter a upper bound of POS (a positive integer):");
    System.out.print(">");
    POS_upper_bound = Integer.parseInt(sc.nextLine());
    if (POS_upper_bound < 1) {
        System.out.println("*** The upper bound entered was invalid ***");
        continue;
    }
}

try (NumberFormatException e) {
    System.out.println("*** The entered String was not an Integer ***");
    continue;
}

// query vcf
rs = stat.executeQuery("SELECT * FROM vcf WHERE vcf ID BETWEEN " + vcf_ID_lower_bound + " AND vcf_ID_upper_bound + " AND POS BETWEEN " + POS_lower_bound + " AND POS_upper_bound + ");

cnt = 0;
ResultSetMetaData rsmd = rs.getMetaData();
int columnsNumber = rsmd.getColumnCount();

while (rs.next()) {
    cnt++;
    for (int i = 1; i <= columnsNumber; i++) {
        if (i > 1)
            System.out.print(" , ");
        String columnValue = rs.getString(i);
        System.out.print(rsmd.getColumnName(i) + " = " + columnValue);
    }
    System.out.println();
}
System.out.println("*** " + cnt + " related VCF records were returned above ***");

// close database
sc.close();
conn.close();
}

try (Exception e) {
    e.printStackTrace();
}

// close database
sc.close();
conn.close();
}
References

Drabas, T. (2016). Reading and writing CSV/TSV files with Python. In Practical data analysis cookbook. essay, Packt Publ.

Eclipse Foundation, I. (n.d.). The Community for Open Innovation and Collaboration: The Eclipse Foundation. Eclipse RSS. https://www.eclipse.org/.

Gourlé, H. (n.d.). File Formats. File Formats - Bioinformatics Tutorials. https://www.hadriengourle.com/tutorials/file_formats/.

Guo, P. (2014, July 7). Python Is Now the Most Popular Introductory Teaching Language at Top -U.S. -Universities. ACM. https://cacm.acm.org/blogs/blog-cacm/176450-python-is-now-the-most-popular-introductory-teaching-language-at-top-us-universities/fulltext.

JetBrains S.R.O. (n.d.). PyCharm: the Python IDE for Professional Developers by JetBrains. JetBrains. https://www.jetbrains.com/pycharm/.

Microsoft. (2016, April 14). Visual studio code - code editing. redefined. Visual Studio Code - Code Editing. Redefined. https://code.visualstudio.com/.

MvnRepository. (n.d.). SQLite JDBC. MavenRepository. https://mvnrepository.com/artifact/org.xerial/sqlite-jdbc.

Python Org. (n.d.). Welcome to Python.org. Python.org. https://www.python.org/.

Ronacher, A. (n.d.). Flask Documentation. Welcome to Flask - Flask Documentation (2.0.x). https://flask.palletsprojects.com/en/2.0.x/.

Ronacher, A. (n.d.). Jinja Documentation. Jinja. https://jinja.palletsprojects.com/en/3.0.x/.

RStudio PBC. (2021, June 22). Open source &amp; professional software for data science teams. RStudio. https://www.rstudio.com/.

Shen, C., Du, H., Chen, Z., Lu, H., Zhu, F., Chen, H., Meng, X., Liu, Q., Liu, P., Zheng, L., Li, X., Dong, J., Liang, C., &amp; Wang, T. (2020). The Chromosome-Level Genome Sequence of the Autotetraploid Alfalfa and Resequencing of Core Germplasms Provide Genomic Resources for Alfalfa Research. Molecular Plant, 13(9), 1250–1261. https://doi.org/10.1016/j.molp.2020.07.003.

SQLite Org. (n.d.). SQLite Home Page. https://www.sqlite.org/index.html.

Sqlitebrowser Org. (2021, May 18). DB Browser for SQLite. http://www.sqlitebrowser.org/.

The Apache Software Foundation. (n.d.). Apache ECharts - An Open Source JavaScript Visualization Library. Apache echarts. https://echarts.apache.org/en/index.html.

The Ensembl Project. (n.d.). GFF/GTF File Format - Definition and supported options. https://useast.ensembl.org/info/website/upload/gff.html.

The Oracle Corporation. (n.d.). Java SE Downloads. Oracle. https://www.oracle.com/java/technologies/javase-downloads.html.

The R Foundation. (n.d.). The R Project for Statistical Computing. R. https://www.r-project.org/.

U.S. National Library of Medicine. (n.d.). BLAST: Basic Local Alignment Search Tool. National Center for Biotechnology Information. https://blast.ncbi.nlm.nih.gov/Blast.cgi.