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

Shichen Qiao^{1,*}, 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. Several 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.

```
(base) [shenchen@wtlab ms1]$ tree -h .
.
|-- [ 53M] AD_rep.R2.gff
|-- [ 11M] alltpm.txt
|-- [779M] ms.main.fasta
|-- [ 26K] ms.main.fasta.fai
|-- [ 58M] MuXu_Last.gff
|-- [4.0K] single_anno
|   |-- [ 48M] MuXu_Last.annotate
|   |-- [1.1M] MuXu_Last.annotate.go
|   |-- [285K] MuXu_Last.annotate.kegg_pathway
|   |-- [1.0M] MuXu_Last.annotate.pathway
|   |-- [1022K] MuXu_Last.annotate.pfam
|   |-- [ 61M] MuXu_Last.cds.fasta
|   |-- [ 53M] MuXu_Last.cds.long.fasta
|   |-- [149M] MuXu_Last.gene.fasta
|   |-- [ 58M] MuXu_Last.gff
|   |-- [4.8M] MuXu_Last.interpro
|   |-- [ 21M] MuXu_Last.pros.fasta
|   |-- [ 18M] MuXu_Last.pros.long.fasta
|   |-- [ 136] test
|-- [113M] single_anno.zip
-- [ 487] test.sh

1 directory, 20 files
(base) [shenchen@wtlab ms1]$
```

```
(base) [shenchen@wtlab vcf]$ tree -h .
.
|-- [528G] all.vcf.bak
|-- [224G] all.vcf.sql.bak
|-- [500M] Chr1.vcf.filtered.500M.vcf
|-- [ 70G] Chr1.vcf.filtered.vcf
|-- [ 20G] Chr1.vcf.filtered.vcf.gz
|-- [101K] Chr1.vcf.filtered.vcf.gz.tbi
|-- [ 63G] Chr2.vcf.filtered.vcf
|-- [ 18G] Chr2.vcf.filtered.vcf.gz
|-- [ 85K] Chr2.vcf.filtered.vcf.gz.tbi
|-- [ 78G] Chr3.vcf.filtered.vcf
|-- [ 22G] Chr3.vcf.filtered.vcf.gz
|-- [104K] Chr3.vcf.filtered.vcf.gz.tbi
|-- [ 70G] Chr4.vcf.filtered.vcf
|-- [ 20G] Chr4.vcf.filtered.vcf.gz
|-- [ 93K] Chr4.vcf.filtered.vcf.gz.tbi
|-- [ 78G] Chr5.vcf.filtered.vcf
|-- [ 18G] Chr5.vcf.filtered.vcf.gz
|-- [ 94K] Chr5.vcf.filtered.vcf.gz.tbi
|-- [ 93G] Chr6.vcf.filtered.vcf
|-- [ 26G] Chr6.vcf.filtered.vcf.gz
|-- [114K] Chr6.vcf.filtered.vcf.gz.tbi
|-- [ 75G] Chr7.vcf.filtered.vcf
|-- [ 21G] Chr7.vcf.filtered.vcf.gz
|-- [ 95K] Chr7.vcf.filtered.vcf.gz.tbi
|-- [ 65G] Chr8.vcf.filtered.vcf
|-- [ 65G] Chr8.vcf.filtered.vcf.bak
|-- [ 90K] Chr8.vcf.filtered.vcf.gz.tbi
|-- [ 27G] chr8.vcf.sql
```

Figure 1. Complete dataset on our CentOS server.

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 base 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**.

```

selected_data — -bash — 60x18
Justin-QiaodeMacBook-Pro:selected_data JustinQiao$ tree -h .
.
├── [500M] Chr1.vcf.filtered.500M.vcf
├── [779M] Ms_genome.fasta
├── [ 48M] MuXu_Last.annotate
├── [1.1M] MuXu_Last.annotate.go
├── [285K] MuXu_Last.annotate.kegg_pathway
├── [1.0M] MuXu_Last.annotate.pathway
├── [1022K] MuXu_Last.annotate.pfam
├── [ 58M] MuXu_Last.gff
├── [4.8M] MuXu_Last.interpro
├── [ 21M] MuXu_Last.pros.fasta
└── [ 11M] alltpm.txt

0 directories, 11 files
Justin-QiaodeMacBook-Pro:selected_data JustinQiao$

```

Figure 2. Selected dataset.

*.vcf							
## (headers...)							
... ..							
## (headers ...)							
CHROM	POS	REF	ALT	C1	C10	C100	... (160 more columns in ASCII order) ...
Chr1	12	G	A, C, T	G/G: ...	./.	A/C: ...	...
Chr1	98	TA	T	TA/T: ...	T/T: ...	./.	...
...	...	...	...	...	...	...	...

.annotate.		AND	*.interpro	
MsG0000000010.01.T01	...		KEGG: 00970+6.1.1.6	...
MsG0000000115.01.T01	...		KEGG: 00230+2.4.2.14	KEGG: 00250+2.4.2.14
MsG0000000120.01.T01	...		...(Tab separated, could be empty)...	...
...(more Gene ID)...	...		...(number of entries on each row varies)	...
...	...		...(could also be data like GO:0003684, PF10604, Alpha/Beta hydrolase fold(IPR029058), etc.)...	...

*.annotate							
MsG1A10005445.01.T01	...	...(annotations)...	...	Gene3D	...	Zinc finger, CCHC-type	... (info2)...
...(Gene ID)...	...	...	...	...(db name)...	...	...(info1)...	... (15 cols in total)...
MsG8A10044948.01.T01	...	...	...	Pfam	...	Heat shock factor	...
...(Could have duplicates)...	...	...	...	...	...	...	...

*.gff							
Chr1	maker	gene	146833	150094	.	+	ID=MsG1A10000001.01;Name=...
...(seqid)...	...(source)...	...(type)...	...(start)...	...(end)...	...(score)...	...(strand)...	...(phase)...
Chr3	maker	mRNA	...	...	591	-	2
...	...	...	...	...	...	...	...
Chr8	...	CDS	...	...	...	...	...

alltpm.txt			
GeneID	150_trueleaf_rep1	150_xia_rep1	...(35 more columns)...
MsG1A10000001.01	60.6945	81.4111	...
MsG1A10000002.01	26.7687	41.1521	...
...(48807 IDs in total)	...	...	...

*.fasta	
>Chr1	TTTACTCGTTTCAACAGA...
>Chr2	TGACGTAAGCTGTACGTAT...
>...	...
>contig1end	ATCGTAGCTATGATCGTAG...
>...	...
>contig649end	ATTTGACGTAGTGTCTTC...

Figure 3. Selected datafiles' formats.

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.

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.

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q
1	MgiG1A10005445.01.T01	720defb494a2f519562	297	Coils	Coil			157	177	-	T	07-08-2019					
2	MgiG1A10005445.01.T01	720defb494a2f519562	297	Gene3D	G3DSA:1.10.10.10			10	103	7.40E-26 T		07-08-2019	IPR038388	Winged helix-like DNA-binding domain superfamily			
3	MgiG1A10005445.01.T01	720defb494a2f519562	297	PRINTS	PRO00056	Heat shock factor (HSF) domain sign		57	69	5.30E-11 T		07-08-2019	IPR000232	Heat shock factor (HSF)-type, DNA-binding	GO:0003700 GO:0005634 GO:0003555 GO:0043565		
4	MgiG1A10005445.01.T01	720defb494a2f519562	297	PRINTS	PRO00056	Heat shock factor (HSF) domain sign		19	42	5.30E-11 T		07-08-2019	IPR000232	Heat shock factor (HSF)-type, DNA-binding	GO:0003700 GO:0005634 GO:0003555 GO:0043565		
5	MgiG1A10005445.01.T01	720defb494a2f519562	297	SUPERFAM	SSF48785			15	101	9.11E-24 T		07-08-2019	IPR038389	Winged helix DNA-binding domain superfamily			
6	MgiG1A10005445.01.T01	720defb494a2f519562	297	SMART	SM00415			15	101	1.20E-42 T		07-08-2019	IPR000232	Heat shock factor (HSF)-type, DNA-binding	GO:0003700 GO:0005634 GO:0003555 GO:0043565		
7	MgiG1A10005445.01.T01	720defb494a2f519562	297	MobiDBLite	mobidb-lite consensus disorder prediction			129	148	-	T	07-08-2019					
8	MgiG1A10005445.01.T01	720defb494a2f519562	297	Pfam	PF00447	HSF-type DNA-binding		19	101	1.40E-19 T		07-08-2019	IPR000232	Heat shock factor (HSF)-type, DNA-binding	GO:0003700 GO:0005634 GO:0003555 GO:0043565		
9	MgiG1A10044948.01.T01	b0ea038a0efb4747171	626	Gene3D	G3DSA:1.10.02.10			274	324	1.90E-06 T		07-08-2019					
10	MgiG1A10044948.01.T01	b0ea038a0efb4747171	626	SUPERFAM	SSF57566			268	302	1.09E-07 T		07-08-2019	IPR038875	Zinc finger, CCHC-type superfamily	GO:0003676 GO:0008270		
11	MgiG1A10044948.01.T01	b0ea038a0efb4747171	626	ProSiteProfi	PSS0158	Zinc finger CCHC-type profile		282	297	9.257 T		07-08-2019	IPR001878	Zinc finger, CCHC-type	GO:0003676 GO:0008270		
12	MgiG1A10044948.01.T01	b0ea038a0efb4747171	626	MobiDBLite	mobidb-lite consensus disorder prediction			299	319	-	T	07-08-2019					
13	MgiG1A10044948.01.T01	b0ea038a0efb4747171	626	Pfam	PF00098	Zinc knuckle		281	296	9.10E-05 T		07-08-2019	IPR001878	Zinc finger, CCHC-type	GO:0003676 GO:0008270		
14	MgiG1A10044948.01.T01	b0ea038a0efb4747171	626	MobiDBLite	mobidb-lite consensus disorder prediction			1	29	-	T	07-08-2019					
15	MgiG1A10044948.01.T01	b0ea038a0efb4747171	626	MobiDBLite	mobidb-lite consensus disorder prediction			456	488	-	T	07-08-2019					
16	MgiG1A10044948.01.T01	b0ea038a0efb4747171	626	MobiDBLite	mobidb-lite consensus disorder prediction			358	378	-	T	07-08-2019					
17	MgiG1A10044948.01.T01	b0ea038a0efb4747171	626	SMART	SM00343			281	297	0.002 T		07-08-2019	IPR001878	Zinc finger, CCHC-type	GO:0003676 GO:0008270		
18	MgiG1A10044948.01.T01	b0ea038a0efb4747171	626	MobiDBLite	mobidb-lite consensus disorder prediction			202	244	-	T	07-08-2019					
19	MgiG1A10038721.01.T01	56d703521d0bf8eb638	226	SUPERFAM	SSF47639			79	219	1.91E-38 T		07-08-2019	IPR038282	Glutathione S-transferase, C-terminal domain superfamily			
20	MgiG1A10038721.01.T01	56d703521d0bf8eb638	226	Gene3D	G3DSA:1.20.1050.10			87	225	6.30E-46 T		07-08-2019					
21	MgiG1A10038721.01.T01	56d703521d0bf8eb638	226	CDD	c033058	GST_N_Tau		8	81	1.14E-41 T		07-08-2019					
22	MgiG1A10038721.01.T01	56d703521d0bf8eb638	226	Coils	Coil			122	142	-	T	07-08-2019					
23	MgiG1A10038721.01.T01	56d703521d0bf8eb638	226	SFLD	SFLD00019	Glutathione Transferase (cytosolic)		12	195	3.70E-35 T		07-08-2019					
24	MgiG1A10038721.01.T01	56d703521d0bf8eb638	226	SFLD	SFLD00056	Main (cytGST)		9	194	2.50E-39 T		07-08-2019					
25	MgiG1A10038721.01.T01	56d703521d0bf8eb638	226	SFLD	SFLD00135	Main 3; Omega- and Tau-like		4	224	0 T		07-08-2019					
26	MgiG1A10038721.01.T01	56d703521d0bf8eb638	226	ProSiteProfi	PSS0045	Soluble glutathione S-transferase C-		90	218	20.90E-7 T		07-08-2019	IPR010987	Glutathione S-transferase, C-terminal-like			
27	MgiG1A10038721.01.T01	56d703521d0bf8eb638	226	Pfam	PF00043	Glutathione S-transferase, C-termini		119	198	1.90E-14 T		07-08-2019	IPR004046	Glutathione S-transferase, C-terminal			
28	MgiG1A10038721.01.T01	56d703521d0bf8eb638	226	Gene3D	G3DSA:2.40.30.10			4	86	8.40E-37 T		07-08-2019					
29	MgiG1A10038721.01.T01	56d703521d0bf8eb638	226	ProSiteProfi	PSS0044	Soluble glutathione S-transferase N		6	85	18.68E-7 T		07-08-2019	IPR004045	Glutathione S-transferase, N-terminal	GO:0005515		
30	MgiG1A10038721.01.T01	56d703521d0bf8eb638	226	CDD	c03185	GST_C_Tau		91	216	3.33E-48 T		07-08-2019					
31	MgiG1A10038721.01.T01	56d703521d0bf8eb638	226	SUPERFAM	SSF52633			7	108	1.36E-22 T		07-08-2019	IPR038249	Thioredoxin-like superfamily			
32	MgiG1A10038721.01.T01	56d703521d0bf8eb638	226	Pfam	PF02796	Glutathione S-transferase, N-termin		7	79	3.20E-19 T		07-08-2019	IPR004045	Glutathione S-transferase, N-terminal	GO:0005515		
33	MgiG1A10000234.01.T01	234978fb4ebd2773a9f1	173	ProSiteProfi	PSS1450	Leucine-rich repeat profile		70	90	5.24E-7 T		07-08-2019	IPR001611	Leucine-rich repeat	GO:0005515		
34	MgiG1A10000234.01.T01	234978fb4ebd2773a9f1	173	ProSiteProfi	PSS1450	Leucine-rich repeat profile		47	68	7.92 T		07-08-2019	IPR001611	Leucine-rich repeat	GO:0005515		
35	MgiG1A10000234.01.T01	234978fb4ebd2773a9f1	173	Pfam	PF13855	Leucine-rich repeat		45	81	1.80E-06 T		07-08-2019	IPR001611	Leucine-rich repeat	GO:0005515		
36	MgiG1A10000234.01.T01	234978fb4ebd2773a9f1	173	SUPERFAM	SSF52058			2	90	1.53E-17 T		07-08-2019					
37	MgiG1A10000234.01.T01	234978fb4ebd2773a9f1	173	Gene3D	G3DSA:3.80.10.10			1	102	2.00E-23 T		07-08-2019	IPR032675	Leucine-rich repeat domain superfamily			
38	MgiG1A10000234.01.T01	234978fb4ebd2773a9f1	173	ProSiteProfi	PSS1450	Leucine-rich repeat profile		1	21	4.803 T		07-08-2019	IPR001611	Leucine-rich repeat	GO:0005515		
39	MgiG1A10000234.01.T01	234978fb4ebd2773a9f1	173	ProSiteProfi	PSS1450	Leucine-rich repeat profile		22	45	5.933 T		07-08-2019	IPR001611	Leucine-rich repeat	GO:0005515		
40	MgiG1A10014106.01.T01	d6d11638a3af6a76b6e6	694	ProSiteProfi	PSS1192	Superfamilies 1 and 2 helicase ATP- γ		65	234	21.493 T		07-08-2019	IPR014001	Helicase superfamily 1/2, ATP-binding domain			

Figure 5. Part of MuXu_Last.annotate shown in Excel.

4.2 Pre-processing Annotation and FASTA Datafiles with Python

After table creations, the raw data needed to be pre-processed before being inserted 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:

```

1. import csv
2.
3. with open("go_code.csv", 'r') as fp:
4.     reader = csv.reader(fp)
5.     rp = open("concatenated_go.csv", 'w')
6.     content = csv.writer(rp)
7.     for x in reader:
8.         temp = ""
9.         for letter in x[0]:
10.            if letter == '\t':
11.                temp = temp + ';'
12.            else:
13.                temp = temp + letter
14.            temp = temp.rstrip(",")
15.            content.writerow([temp])
16. fp.close()
17. 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:

```

1. import csv
2.
3. def readFa(fa):
4.     with open(fa,'r') as FA:
5.         seqName,seq="",
6.         while 1:
7.             line=FA.readline()
8.             line=line.strip("\n")
9.             if (line.startswith('>') or not line) and seqName:
10.                 yield((seqName,seq))
11.             if line.startswith('>'):
12.                 seqName = line[1:]
13.                 seq=""
14.             else:
15.                 seq+=line
16.             if not line:break
17.
18. with open('protein_sequence.csv', 'w') as fp:
19.     content = csv.writer(fp)
20.     fa = "/MuXu_Last.pros.fasta"
21.     for seqName, seq in readFa(fa):
22.         content.writerow([seqName, seq])
23.     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**).

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
1		1 Chr1	maker	gene	146833	150094		+	.	ID=MsG1A10000001.01;Name=MsG1A10000001.01					
2		2 Chr1	maker	mRNA	146833	150094		+	.	ID=MsG1A10000001.01.T01;Parent=MsG1A10000001.01;Name=MsG1A10					
3		3 Chr1	maker	exon	149391	150094		+	.	ID=MsG1A10000001.01.T01.exon8067;Parent=MsG1A10000001.01.T01					
4		4 Chr1	maker	exon	149113	149274		+	.	ID=MsG1A10000001.01.T01.exon8066;Parent=MsG1A10000001.01.T01					
5		5 Chr1	maker	exon	148207	148445		+	.	ID=MsG1A10000001.01.T01.exon8065;Parent=MsG1A10000001.01.T01					
6		6 Chr1	maker	exon	147890	148028		+	.	ID=MsG1A10000001.01.T01.exon8064;Parent=MsG1A10000001.01.T01					
7		7 Chr1	maker	exon	147592	147827		+	.	ID=MsG1A10000001.01.T01.exon8063;Parent=MsG1A10000001.01.T01					
8		8 Chr1	maker	exon	146833	147471		+	.	ID=MsG1A10000001.01.T01.exon8062;Parent=MsG1A10000001.01.T01					
9		9 Chr1	maker	five_prime_U	146833	147341		+	.	ID=MsG1A10000001.01.T01.five_prime utr;Parent=MsG1A10000001.01.T01					
10		10 Chr1	maker	CDS	147342	147471		+	.	0 ID=MsG1A10000001.01.T01.cds;Parent=MsG1A10000001.01.T01					
11		11 Chr1	maker	CDS	147592	147827		+	.	2 ID=MsG1A10000001.01.T01.cds;Parent=MsG1A10000001.01.T01					
12		12 Chr1	maker	CDS	147890	148028		+	.	0 ID=MsG1A10000001.01.T01.cds;Parent=MsG1A10000001.01.T01					
13		13 Chr1	maker	CDS	148207	148445		+	.	2 ID=MsG1A10000001.01.T01.cds;Parent=MsG1A10000001.01.T01					
14		14 Chr1	maker	CDS	149113	149274		+	.	0 ID=MsG1A10000001.01.T01.cds;Parent=MsG1A10000001.01.T01					
15		15 Chr1	maker	CDS	149391	149750		+	.	0 ID=MsG1A10000001.01.T01.cds;Parent=MsG1A10000001.01.T01					
16		16 Chr1	maker	three_prime	149751	150094		+	.	ID=MsG1A10000001.01.T01.three_prime utr;Parent=MsG1A10000001.01.T01					
17		17 Chr1	maker	gene	152749	157114		+	.	ID=MsG1A10000002.01;Name=MsG1A10000002.01					
18		18 Chr1	maker	mRNA	152749	157114		+	.	ID=MsG1A10000002.01.T01;Parent=MsG1A10000002.01;Name=MsG1A10					
19		19 Chr1	maker	exon	156584	157114		+	.	ID=MsG1A10000002.01.T01.exon8061;Parent=MsG1A10000002.01.T01					
20		20 Chr1	maker	exon	155015	156197		+	.	ID=MsG1A10000002.01.T01.exon8060;Parent=MsG1A10000002.01.T01					
21		21 Chr1	maker	exon	152749	153142		+	.	ID=MsG1A10000002.01.T01.exon8059;Parent=MsG1A10000002.01.T01					
22		22 Chr1	maker	five_prime_U	152749	153103		+	.	ID=MsG1A10000002.01.T01.five_prime utr;Parent=MsG1A10000002.01.T01					
23		23 Chr1	maker	CDS	153104	153142		+	.	0 ID=MsG1A10000002.01.T01.cds;Parent=MsG1A10000002.01.T01					
24		24 Chr1	maker	CDS	155015	156197		+	.	0 ID=MsG1A10000002.01.T01.cds;Parent=MsG1A10000002.01.T01					
25		25 Chr1	maker	CDS	156584	156783		+	.	2 ID=MsG1A10000002.01.T01.cds;Parent=MsG1A10000002.01.T01					
26		26 Chr1	maker	three_prime	156784	157114		+	.	ID=MsG1A10000002.01.T01.three_prime utr;Parent=MsG1A10000002.01.T01					
27		27 Chr1	maker	gene	159737	169533		-	.	ID=MsG1A10000003.01;Name=MsG1A10000003.01					
28		28 Chr1	maker	mRNA	159737	169533		-	.	ID=MsG1A10000003.01.T01;Parent=MsG1A10000003.01;Name=MsG1A10					
29		29 Chr1	maker	exon	169267	169533		-	.	ID=MsG1A10000003.01.T01.exon8033;Parent=MsG1A10000003.01.T01					
30		30 Chr1	maker	exon	168714	169056		-	.	ID=MsG1A10000003.01.T01.exon8034;Parent=MsG1A10000003.01.T01					
31		31 Chr1	maker	exon	167903	168174		-	.	ID=MsG1A10000003.01.T01.exon8035;Parent=MsG1A10000003.01.T01					
32		32 Chr1	maker	exon	167629	167828		-	.	ID=MsG1A10000003.01.T01.exon8036;Parent=MsG1A10000003.01.T01					
33		33 Chr1	maker	exon	166595	166962		-	.	ID=MsG1A10000003.01.T01.exon8037;Parent=MsG1A10000003.01.T01					
34		34 Chr1	maker	exon	163309	163656		-	.	ID=MsG1A10000003.01.T01.exon8038;Parent=MsG1A10000003.01.T01					
35		35 Chr1	maker	exon	163066	163215		-	.	ID=MsG1A10000003.01.T01.exon8039;Parent=MsG1A10000003.01.T01					

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, source, 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, source, type, start, end, score, strand, phase, attributes) VALUES("1", "Chr1", "maker", "gene", "146833", "150094", ".", "+", ".", "ID=MsG1A10000001.01;Name=MsG1A10000001.01");

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**.

	unique_id	seqid	source	type	start	end	score	strand	phase	attributes
	Filter	Filter	Filter	Filter	Filter	Filter	Filter	Filter	Filter	Filter
1		1	Chr1	maker	gene	146833	150094	.	+	ID=MsG1A10000001.01;Name=MsG1A10000001.01
2		2	Chr1	maker	mRNA	146833	150094	.	+	ID=MsG1A10000001.01.T01;Parent=MsG1A10000001.01;Name=MsG1...
3		3	Chr1	maker	exon	149391	150094	.	+	ID=MsG1A10000001.01.T01:exon:8067;Parent=MsG1A10000001.01.T01
4		4	Chr1	maker	exon	149113	149274	.	+	ID=MsG1A10000001.01.T01:exon:8066;Parent=MsG1A10000001.01.T01
5		5	Chr1	maker	exon	148207	148445	.	+	ID=MsG1A10000001.01.T01:exon:8065;Parent=MsG1A10000001.01.T01
6		6	Chr1	maker	exon	147890	148028	.	+	ID=MsG1A10000001.01.T01:exon:8064;Parent=MsG1A10000001.01.T01
7		7	Chr1	maker	exon	147592	147827	.	+	ID=MsG1A10000001.01.T01:exon:8063;Parent=MsG1A10000001.01.T01
8		8	Chr1	maker	exon	146833	147471	.	+	ID=MsG1A10000001.01.T01:exon:8062;Parent=MsG1A10000001.01.T01
9		9	Chr1	maker	five_prime_UTR	146833	147341	.	+	ID=MsG1A10000001.01.T01:five_prime utr;Parent=MsG1A10000001.01...
10		10	Chr1	maker	CDS	147342	147471	.	0	ID=MsG1A10000001.01.T01:cds;Parent=MsG1A10000001.01.T01
11		11	Chr1	maker	CDS	147592	147827	.	2	ID=MsG1A10000001.01.T01:cds;Parent=MsG1A10000001.01.T01
12		12	Chr1	maker	CDS	147890	148028	.	0	ID=MsG1A10000001.01.T01:cds;Parent=MsG1A10000001.01.T01
13		13	Chr1	maker	CDS	148207	148445	.	2	ID=MsG1A10000001.01.T01:cds;Parent=MsG1A10000001.01.T01
14		14	Chr1	maker	CDS	149113	149274	.	0	ID=MsG1A10000001.01.T01:cds;Parent=MsG1A10000001.01.T01
15		15	Chr1	maker	CDS	149391	149750	.	0	ID=MsG1A10000001.01.T01:cds;Parent=MsG1A10000001.01.T01
16		16	Chr1	maker	three_prime_UTR	149751	150094	.	+	ID=MsG1A10000001.01.T01:three_prime utr;Parent=MsG1A10000001.0...
17		17	Chr1	maker	gene	152749	157114	.	+	ID=MsG1A10000002.01;Name=MsG1A10000002.01
18		18	Chr1	maker	mRNA	152749	157114	.	+	ID=MsG1A10000002.01.T01;Parent=MsG1A10000002.01;Name=MsG1...
19		19	Chr1	maker	exon	156584	157114	.	+	ID=MsG1A10000002.01.T01:exon:8061;Parent=MsG1A10000002.01.T01
20		20	Chr1	maker	exon	155015	156197	.	+	ID=MsG1A10000002.01.T01:exon:8060;Parent=MsG1A10000002.01.T01
21		21	Chr1	maker	exon	152749	153142	.	+	ID=MsG1A10000002.01.T01:exon:8059;Parent=MsG1A10000002.01.T01
22		22	Chr1	maker	five_prime_UTR	152749	153103	.	+	ID=MsG1A10000002.01.T01:five_prime utr;Parent=MsG1A10000002.01...
23		23	Chr1	maker	CDS	153104	153142	.	0	ID=MsG1A10000002.01.T01:cds;Parent=MsG1A10000002.01.T01
24		24	Chr1	maker	CDS	155015	156197	.	0	ID=MsG1A10000002.01.T01:cds;Parent=MsG1A10000002.01.T01
25		25	Chr1	maker	CDS	156584	156783	.	2	ID=MsG1A10000002.01.T01:cds;Parent=MsG1A10000002.01.T01
26		26	Chr1	maker	three_prime_UTR	156784	157114	.	+	ID=MsG1A10000002.01.T01:three_prime utr;Parent=MsG1A10000002.0...
27		27	Chr1	maker	gene	159737	169533	.	-	ID=MsG1A10000003.01;Name=MsG1A10000003.01
28		28	Chr1	maker	mRNA	159737	169533	.	-	ID=MsG1A10000003.01.T01;Parent=MsG1A10000003.01;Name=MsG1...
29		29	Chr1	maker	exon	169267	169533	.	-	ID=MsG1A10000003.01.T01:exon:8033;Parent=MsG1A10000003.01.T01
30		30	Chr1	maker	exon	168714	169056	.	-	ID=MsG1A10000003.01.T01:exon:8034;Parent=MsG1A10000003.01.T01

Figure 7. Part of Table gff in Alfalfa.db shown in DB Browser for SQLite.

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. **INSERT INTO** annotate_info(ID_full,db_name,info1,info2) **VALUES**("MsG1A10000031.01.T01","ProSitePatterns", "Lipolytic enzymes "G-D-S-L" family, serine active site.", "Lipase, GDSL, active site");

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. **import** csv
- 2.
3. with open('alltmp.csv', 'r') as fp:
4. reader = csv.reader(fp)
5. rf = open('alltmp.sql', 'w')

```

6.     for row in reader:
7.         values = ""
8.         for x in row:
9.             values += "\"" + x + "\", "
10.        values = values[0 : len(values) - 1]
11.        sql = "INSERT INTO alltpm(GeneID, \"150_trueleaf_rep1\", \"150_xia_rep1\", \"150_ziye_rep1\", fennie_rep1, fl
ower bud rep1, flower rep1, flower rep2, flower rep3, flower rep4, flower rep5, flower rep6, flower rep7, flower r
ep8, 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_rep
1, nod_28_rep1, seed_rep1, stem_branch_rep1, stem_rep1, yige_rep1, yige_rep2, yige_rep3, young_leaf_rep1, young_l
eaf_rep2, young leaf rep3) VALUES(" + values + ");\n"
12.        rf.write(sql)
13.        fp.close()
14.        rf.close()

```

4.4 Improving Table gff

As mentioned in Section 4.1, a linkage table called “ID_to_uniqueid” was created to optimize gene ID queries from Table gff. The first 13 characters after “ID=” in the attribute column in Table gff would be extracted out and stored in the ID column in this helper Table. The extraction was done by the Convert Text to Columns Wizard in Excel, using “=” and “.” as delimiters. Then, the unique_id column and the extracted primary ID column were saved as a separate CSV file named “uniqueid_ID_primary.csv” (**Figure 8**).

	A	B	C
1	1	MsG1A10000001	
2	2	MsG1A10000001	
3	3	MsG1A10000001	
4	4	MsG1A10000001	
5	5	MsG1A10000001	
6	6	MsG1A10000001	
7	7	MsG1A10000001	
8	8	MsG1A10000001	
9	9	MsG1A10000001	
10	10	MsG1A10000001	
11	11	MsG1A10000001	
12	12	MsG1A10000001	
13	13	MsG1A10000001	
14	14	MsG1A10000001	
15	15	MsG1A10000001	
16	16	MsG1A10000001	
17	17	MsG1A10000002	
18	18	MsG1A10000002	
19	19	MsG1A10000002	
20	20	MsG1A10000002	
21	21	MsG1A10000002	
22	22	MsG1A10000002	
23	23	MsG1A10000002	
24	24	MsG1A10000002	
25	25	MsG1A10000002	
26	26	MsG1A10000002	
27	27	MsG1A10000003	
28	28	MsG1A10000003	
29	29	MsG1A10000003	
30	30	MsG1A10000003	
31	31	MsG1A10000003	
32	32	MsG1A10000003	

Figure 8. Part of uniqueid_ID_primary.csv shown in Excel.

Next, the following Python script was executed on this CSV file to find out the starting and ending unique_id matched to different primary IDs:

```

1. import csv
2.
3. with open('uniqueid ID primary.csv', 'r') as fp:
4.     reader = csv.reader(fp)
5.     rp = open('linkage_ID_to_uniqueid.csv', 'w')
6.     content = csv.writer(rp)
7.     temp = -1
8.     start = 1
9.     end = 1
10.    for x in reader:
11.        if x[0] == 1:
12.            name = x[1]
13.            if x[1] != temp:
14.                if temp != -1:
15.                    print(name, 'starts from', start, 'ends at', end - 1)
16.                    content.writerow([name, start, end - 1])
17.                name = x[1]
18.                start = int(x[0])
19.                temp = x[1]
20.                end = end + 1
21.            content.writerow([x[1], start, x[0]])
22.    fp.close()
23.    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:

```

1. =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 ""&A1&"" 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).

```

1. fa="/Ms_genome.fasta"
2. for seqName,seq in readFa(fa):
3.     seqLen = len(seq)

```



```

4.   fileName = seqName + ".txt"
5.   file = open(fileName, 'w');
6.   file.write(seq)
7.   file.close()
8.   print(seqName, seqLen)

```

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:

```

1.  import csv
2.
3.  with open('start_end_raw.csv', 'r') as fp:
4.      reader = csv.reader(fp)
5.      rf = open('sequences.sql', 'w')
6.      f = open('Chr1.txt', 'r')
7.      for x in reader:
8.          unique_id = x[0]
9.          seqName = x[1]
10.         start = x[2]
11.         end = x[3]
12.         fileName = seqName + '.txt'
13.         f = open(fileName, 'r')
14.         data = f.read()
15.         f.close()
16.         seq = data[int(start) - 1:int(end)]
17.         sql = "UPDATE gff SET DNA = '" + seq + "' WHERE unique_id = '" + unique_id + "';\n"
18.         rf.write(sql)
19.     fp.close()
20.     rf.close()

```

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

	unique_id	seqid	source	type	start	end	score	strand	phase	attributes	DNA
1	1	Chr1	maker	gene	146833	150094	.	+	.	ID=MeG1A10000001.01;Name=MeG1A10000001.01	ACCGACTCTATATTCCTTTTCGGTGACGAACAAAG...
2	2	Chr1	maker	mRNA	146833	150094	.	+	.	ID=MeG1A10000001.01.T01;Parent=MeG1A10000001.01;Na...	ACCGACTCTATATTCCTTTTCGGTGACGAACAAAG...
3	3	Chr1	maker	exon	146831	150094	.	+	.	ID=MeG1A10000001.01.T01.exon...	GTTTTCATCTCTTCATGACGAGTTCGTTGTTGTAAT...
4	4	Chr1	maker	exon	149113	149274	.	+	.	ID=MeG1A10000001.01.T01.exon...	TCAATCGGTTCTCTACAAAAGAAATTCGATTGGATCTTT...
5	5	Chr1	maker	exon	148207	148445	.	+	.	ID=MeG1A10000001.01.T01.exon...	TTTAAATTTTCATTTCTCATCGTGTATGTGAACAGATATC...
6	6	Chr1	maker	exon	147890	148028	.	+	.	ID=MeG1A10000001.01.T01.exon...	ATCACTACAGCCAGTGGAAACCTGGGAATCTATAAGAGA...
7	7	Chr1	maker	exon	147892	147827	.	+	.	ID=MeG1A10000001.01.T01.exon...	CGACTACTTTGACTGGGTTTCACTTTCTGTAACGCTCTT...
8	8	Chr1	maker	exon	146833	147471	.	+	.	ID=MeG1A10000001.01.T01.exon...	ACCGACTCTATATTCCTTTTCGGTGACGAACAAAG...
9	9	Chr1	maker	five_prime_UTR	146833	147341	.	+	.	ID=MeG1A10000001.01.T01.five_prime_utr;Parent=MeG1A10...	ACCGACTCTATATTCCTTTTCGGTGACGAACAAAG...
10	10	Chr1	maker	CDS	147342	147471	.	+	0	ID=MeG1A10000001.01.T01.cds;Parent=MeG1A10000001.01...	ATGGAGAGTGAAAGAAATCGTCGCAATATCGAAGCTTTG...
11	11	Chr1	maker	CDS	147892	147827	.	+	2	ID=MeG1A10000001.01.T01.cds;Parent=MeG1A10000001.01...	CGACTACTTTGACTGGGTTTCACTTTCTGTAACGCTCTT...
12	12	Chr1	maker	CDS	147890	148028	.	+	0	ID=MeG1A10000001.01.T01.cds;Parent=MeG1A10000001.01...	ATCACTACAGCCAGTGGAAACCTGGGAATCTATAAGAGA...
13	13	Chr1	maker	CDS	148207	148445	.	+	2	ID=MeG1A10000001.01.T01.cds;Parent=MeG1A10000001.01...	TTTAAATTTTCATTTCTCATCGTGTATGTGAACAGATATC...
14	14	Chr1	maker	CDS	149113	149274	.	+	0	ID=MeG1A10000001.01.T01.cds;Parent=MeG1A10000001.01...	TCAATCGGTTCTCTACAAAAGAAATTCGATTGGATCTTT...
15	15	Chr1	maker	CDS	149391	149750	.	+	0	ID=MeG1A10000001.01.T01.cds;Parent=MeG1A10000001.01...	GTTTTCATCTCTTCATGACGATCTAGCTGTTTGTGAAT...
16	16	Chr1	maker	three_prime_UTR	149751	150094	.	+	.	ID=MeG1A10000001.01.T01.three_prime_utr;Parent=MeG1A1...	AATGATCTCACTCACACAGATGGCAAAATATAAATGT...
17	17	Chr1	maker	gene	152749	157114	.	+	.	ID=MeG1A10000002.01;Name=MeG1A10000002.01	GTGTGTGACTTATCTGTATCTATCTTGGTCGACCTCAAT...
18	18	Chr1	maker	mRNA	152749	157114	.	+	.	ID=MeG1A10000002.01.T01;Parent=MeG1A10000002.01;Na...	GTGTGTGACTTATCTGTATCTATCTTGGTCGACCTCAAT...
19	19	Chr1	maker	exon	156684	157114	.	+	.	ID=MeG1A10000002.01.T01.exon...	ATCAAAATCACCGGATATTTAGCAACAGCTCTGGATTA...
20	20	Chr1	maker	exon	155015	156197	.	+	.	ID=MeG1A10000002.01.T01.exon...	AGAAAAGATGGGGAGGTTGTTGGGTCGACTTTCTGTT...
21	21	Chr1	maker	exon	152749	153142	.	+	.	ID=MeG1A10000002.01.T01.exon...	GTGTGTGACTTATCTGTATCTATCTTGGTCGACCTCAAT...
22	22	Chr1	maker	five_prime_UTR	152749	153103	.	+	.	ID=MeG1A10000002.01.T01.five_prime_utr;Parent=MeG1A10...	GTGTGTGACTTATCTGTATCTATCTTGGTCGACCTCAAT...
23	23	Chr1	maker	CDS	153104	153142	.	+	0	ID=MeG1A10000002.01.T01.cds;Parent=MeG1A10000002.01...	ATGGGTCGCAACAAATCGTCCCAAAATGAAGAG...
24	24	Chr1	maker	CDS	155015	156197	.	+	0	ID=MeG1A10000002.01.T01.cds;Parent=MeG1A10000002.01...	AGAAAAGATGGGGAGGTTGTTGGGTCGACTTTCTGTT...
25	25	Chr1	maker	CDS	156684	156783	.	+	2	ID=MeG1A10000002.01.T01.cds;Parent=MeG1A10000002.01...	ATCAAAATCACCGGATATTTAGCAACAGCTCTGGATTA...
26	26	Chr1	maker	three_prime_UTR	156784	157114	.	+	.	ID=MeG1A10000002.01.T01.three_prime_utr;Parent=MeG1A1...	TTGTGATCACTGGGTAATCAATGTGACAGACTGTCG...
27	27	Chr1	maker	gene	159737	169533	.	+	.	ID=MeG1A10000003.01;Name=MeG1A10000003.01	GAAAGTAACTTAATTTGTTTGGAGGCTCTGTTGAAT...
28	28	Chr1	maker	mRNA	159737	169533	.	+	.	ID=MeG1A10000003.01.T01;Parent=MeG1A10000003.01;Na...	GAAAGTAACTTAATTTGTTTGGAGGCTCTGTTGAAT...
29	29	Chr1	maker	exon	169597	169533	.	+	.	ID=MeG1A10000003.01.T01.exon...	CTTGATGACAGTTGACCTCCCTGAGTTAAGAGATGAGAC...
30	30	Chr1	maker	exon	168714	169056	.	+	.	ID=MeG1A10000003.01.T01.exon...	TTGAGATCGGGGATATCTTTAGAACCACTGTTCCAGC...
31	31	Chr1	maker	exon	167903	168174	.	+	.	ID=MeG1A10000003.01.T01.exon...	CGTCACATAAAGATGCTGGCCAGCAGCTATAGTTGT...
32	32	Chr1	maker	exon	147890	147938	.	+	.	ID=MeG1A10000001.01.T01.exon...	CTTGATGCTTCACTTGGGAGGAGGATGATGATGATG...

Figure 9. Finalized Table gff with DNA sequences.

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).

```
column_names.txt — 已編輯
CHROM 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 C125 C126 C127 C128 C129 C13 C130 C131 C132
C133 C134 C135 C136 C137 C14 C15 C16 C17 C18 C19 C2 C20 C21
C22 C23 C24 C25 C26 C27 C28 C29 C3 C30 C31 C32 C33 C34
C35 C36 C37 C38 C39 C4 C40 C41 C42 C43 C44 C45 C46 C47
C48 C49 C5 C50 C51 C52 C53 C54 C55 C56 C57 C58 C59 C6
C60 C61 C62 C63 C64 C65 C66 C67 C68 C69 C7 C70 C71 C72
C73 C74 C75 C76 C77 C78 C79 C8 C80 C81 C82 C83 C84 C85
C86 C87 C88 C89 C9 C90 C91 C92 C93 C94 C95 C96 C97 C98
C99 L1 L10 L11 L12 L13 L15 L16 L17 L18 L19 L2 L20 L21
L22 L27 L3 L32 L33 L4 L5 L6 L7 L8 L9 MsREF zhong lan
```

Figure 10. `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

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**).

vcf_ID	CHROM	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
60	Chr1	795	A	T	J	A/A	J	J	J	A/A	A/A	J	A/A	J	J	J	J	J	J	J	J	J	J	A/A	J	A/A	J	A/A	J	J
61	Chr1	797	A	T	A/A	A/A	J	J	J	A/A	A/A	J	A/A	J	J	J	J	J	J	J	J	J	J	A/A	J	J	J	A/A	J	J
62	Chr1	798	G	A,C,T	A/A	G/G	J	J	J	J	T/T	J	C/C	J	G/G	J	J	J	J	J	J	J	J	J	J	G/G	J	T/T	J	J
63	Chr1	799	C	T	C/C	C/C	J	J	J	J	C/C	J	C/C	J	C/C	J	J	J	J	J	J	J	J	C/C	J	C/C	J	C/C	J	J
64	Chr1	801	A	T	A/A	A/A	J	J	J	J	A/A	A/A	J	J	J	A/A	J	J	J	J	J	J	J	A/A	J	A/A	J	A/A	J	J
65	Chr1	803	G	C	G/G	J	J	J	J	G/G	G/G	J	G/G	J	G/G	J	J	J	J	J	J	J	J	G/G	J	G/G	J	G/C	J	J
66	Chr1	810	A	T	T/T	A/A	J	J	J	J	A/A	A/A	J	J	J	A/A	J	J	J	J	J	A/A	J	A/A	J	J	J	A/A	J	J
67	Chr1	813	G	C	J	J	J	J	J	G/G	G/G	J	J	J	J	J	J	J	J	J	J	J	J	G/G	J	J	J	G/C	J	J
68	Chr1	814	T	G	J	T/T	J	J	J	J	T/T	T/T	J	J	J	J	J	J	J	J	J	T/T	J	T/T	J	J	J	T/T	J	J
69	Chr1	815	T	A	J	J	J	J	J	T/T	T/T	J	J	J	J	J	J	J	J	J	J	T/T	J	T/T	J	J	J	T/T	J	J
70	Chr1	816	T	A	J	T/T	J	J	J	T/T	T/T	J	J	J	J	J	J	J	J	J	J	T/T	J	T/T	J	J	J	T/T	J	J
71	Chr1	817	A	T	A/A	A/A	J	J	J	J	A/A	A/A	J	J	J	J	J	J	J	J	J	A/A	J	A/A	J	J	J	A/A	J	J
72	Chr1	821	T	C	J	T/T	J	J	J	J	T/T	J	J	J	J	J	J	J	J	J	J	T/T	J	T/T	J	J	J	J	J	J
73	Chr1	823	T	A	T	T/T	J	J	J	J	T/T	J	J	J	J	J	J	J	J	J	J	T/T	J	T/T	J	J	J	T/T	J	J
74	Chr1	829	C	T	C/C	C/T	J	J	J	J	T/T	J	J	J	J	J	J	J	J	J	J	C/C	J	C/C	J	C/C	J	C/C	J	J
75	Chr1	831	C	T	C/C	C/C	J	J	J	J	C/C	J	J	J	J	J	J	J	J	J	J	C/C	J	C/C	J	C/C	J	C/C	J	J
76	Chr1	833	T	A	J	T/T	J	J	J	J	T/T	J	J	J	J	J	J	J	J	J	J	J	T/T	J	T/T	J	T/T	J	J	J
77	Chr1	840	C	T	C/C	C/C	J	J	J	C/C	C/C	J	J	J	C/C	J	J	J	J	J	J	C/C	J	C/C	J	C/C	J	C/C	J	J
78	Chr1	842	G	C	G/G	G/G	J	J	J	G/G	G/G	J	J	J	G/G	J	J	J	J	J	J	G/G	J	G/G	J	G/G	J	G/G	J	J
79	Chr1	848	A	T	A/A	A/A	J	J	J	J	A/A	A/A	J	J	A/A	J	J	J	J	J	J	A/A	J	A/A	J	J	J	A/A	J	J
80	Chr1	847	A	T	A/A	A/A	A/A	J	J	A/A	A/A	J	J	A/A	J	J	J	J	J	J	J	A/A	J	A/A	J	J	J	A/A	J	J
81	Chr1	848	A	T	A/A	A/A	A/A	J	J	A/A	A/A	J	J	A/A	J	J	J	J	J	J	J	T/T	J	A/A	J	J	J	A/A	J	J
82	Chr1	850	G	C	G/G	G/G	G/G	J	J	G/G	G/G	J	J	J	J	J	J	J	J	J	J	G/G	J	G/G	J	J	J	G/C	J	J
83	Chr1	853	T	G	J	T/T	J	J	J	J	T/T	J	J	J	J	J	J	J	J	J	J	T/T	J	T/T	J	J	J	T/T	J	J
84	Chr1	854	G	T	G/G	G/G	G/G	J	J	G/G	G/G	J	J	J	G/G	J	J	J	J	J	J	G/G	J	G/G	J	J	J	G/G	J	J
85	Chr1	855	T	G	T/T	J	T/T	J	J	J	T/T	J	J	J	J	J	J	J	J	J	J	T/T	J	T/T	J	J	J	T/T	J	J
86	Chr1	856	T	C	T/T	T/T	T/T	J	J	T/T	T/T	J	J	J	T/T	J	J	J	J	J	J	T/T	J	T/T	J	J	J	T/T	J	J
87	Chr1	858	A	T	A/A	A/A	J	J	J	A/A	A/A	J	J	J	J	J	J	J	J	J	J	A/A	J	A/A	J	J	J	A/A	J	J
88	Chr1	860	A	T	A/A	T/T	A/A	J	J	A/A	A/A	J	J	J	A/A	J	J	J	J	J	J	A/A	J	A/A	J	J	J	A/A	J	J
89	Chr1	862	A	T	J	A/A	A/A	J	J	A/A	A/A	J	J	J	A/A	J	J	J	J	J	J	A/A	J	A/A	J	J	J	A/A	J	J
90	Chr1	864	T	A	J	T/T	A/A	J	J	T/T	T/T	J	J	J	T/T	J	J	J	J	J	J	T/T	J	T/T	J	J	J	T/T	J	J

Figure 12. Part of Table vcf shown in DB Browser for SQLite.

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.

	vcf_ID	CHROM	POS	REF	ALT	C1	C10	C100	C101
1	394	Chr1	4555	A	G	A/G	J.	J.	A/A
2	395	Chr1	4555	A	AG	AG/AG	J.	J.	A/AG
3	527	Chr1	5096	C	T	C/C	C/C	C/C	C/C
4	528	Chr1	5096	CG	C	CG/CG	CG/CG	CG/C	CG/CG
5	529	Chr1	5097	G	C,T	G/C	G/G	G/T	G/G
6	530	Chr1	5097	GT	G	GT/GT	G/G	GT/GT	J.
7	560	Chr1	5209	G	A	J.	J.	J.	J.
8	561	Chr1	5209	G	GA	J.	J.	J.	J.
9	701	Chr1	6612	A	T	A/A	A/A	A/A	J.
10	702	Chr1	6612	A	AT	AT/AT	AT/AT	AT/AT	J.
11	778	Chr1	7558	A	G	A/A	G/G	J.	J.
12	779	Chr1	7558	A	AG	A/AG	AG/AG	J.	J.
13	790	Chr1	8083	G	A	J.	J.	G/G	A/A
14	791	Chr1	8083	G	GA	J.	J.	G/G	GA/GA
15	896	Chr1	9111	A	T	A/A	A/A	J.	J.
16	897	Chr1	9111	A	AT	AT/AT	A/A	J.	J.
17	1045	Chr1	10115	T	A	T/T	J.	T/T	J.

Execution finished without errors.
Result: 5860 rows returned in 5340ms
At line 1:
SELECT * FROM vcf WHERE POS in (SELECT POS FROM vcf GROUP BY POS HAVING count(POS) > 1);

Figure 13. POS problem.

Thus, a jump table was created as follows:

1. CREATE TABLE "vcf_jump_table" (
2. "vcf_ID" INT NOT NULL UNIQUE,
3. "POS" INT NOT NULL,
4. PRIMARY KEY("vcf_ID")
5.);
6. 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.

Table: vcf_jump_tabl

	vcf_ID	POS
	Filter	Filter
1	10000	86768
2	20000	159951
3	30000	230366
4	40000	324488
5	50000	390416
6	60000	439182
7	70000	501325
8	80000	581596
9	90000	637758
10	100000	700244
11	110000	765915
12	120000	826961
13	130000	887818
14	140000	936134
15	150000	1001983
16	160000	1053132
17	170000	1115634
18	180000	1183399
19	190000	1271987
20	200000	1326266
21	210000	1395790
22	220000	1461338
23	230000	1541853
24	240000	1608512
25	250000	1679914

1 - 26 of 13201

Figure 14. Part of Table vcf_jump_table shown in DB Browser for SQLite.

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 Templet 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.

```

Justin-QiaodeMacBook-Pro:alfalfa_database_frontend JustinQiao$ tree -L 3 .
.
├── README.md
├── app.py
├── env
├── bin
│   ├── activate
│   ├── activate.csh
│   ├── activate.fish
│   ├── activate.ps1
│   ├── activate.xsh
│   ├── activate_this.py
│   ├── flask
│   ├── pip
│   ├── pip-3.8
│   ├── pip3
│   ├── pip3.8
│   ├── python -> /Library/Frameworks/Python.framework/Versions/3.8/bin/python3.8
│   ├── python3 -> python
│   ├── python3.8 -> python
│   ├── wheel
│   ├── wheel-3.8
│   ├── wheel3
│   └── wheel3.8
├── include
│   └── site
├── lib
│   └── python3.8
├── pyvenv.cfg
├── static
│   ├── DNA
│   │   ├── Chr1.txt
│   │   ├── Chr2.txt
│   │   ├── Chr3.txt
│   │   ├── Chr4.txt
│   │   ├── Chr5.txt
│   │   ├── Chr6.txt
│   │   ├── Chr7.txt
│   │   └── Chr8.txt
│   ├── css
│   │   └── main.css
│   └── sql
│       ├── Alfalfa.db
│       ├── get_max_vcfID.sql
│       └── get_min_vcfIDs.sql
└── templates
    ├── base.html
    ├── error.html
    └── home.html

11 directories, 36 files
Justin-QiaodeMacBook-Pro:alfalfa_database_frontend JustinQiao$

```

Figure 15. Final project folder of the frontend web application.

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):


```
1. SELECT * FROM annotate_info WHERE info1 LIKE "%HSF%" OR info2 LIKE "%HSF%";
```

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(vcf_ID) FROM vcf;
```

and get_min_vcfIDs.sql:

```
1. SELECT CHROM, min(vcf_ID) FROM vcf WHERE CHROM = "Chr1";
2. SELECT CHROM, min(vcf_ID) FROM vcf WHERE CHROM = "Chr2";
3. SELECT CHROM, min(vcf_ID) FROM vcf WHERE CHROM = "Chr3";
4. SELECT CHROM, min(vcf_ID) FROM vcf WHERE CHROM = "Chr4";
5. SELECT CHROM, min(vcf_ID) FROM vcf WHERE CHROM = "Chr5";
6. SELECT CHROM, min(vcf_ID) FROM vcf WHERE CHROM = "Chr6";
7. SELECT CHROM, min(vcf_ID) FROM vcf WHERE CHROM = "Chr7";
8. SELECT CHROM, min(vcf_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 <vcf_ID_low> AND <vcf_ID_high> AND POS <=
<input_low> ORDER BY vcf_ID DESC LIMIT 1;
2. SELECT * FROM vcf_jump_table WHERE vcf_ID BETWEEN <vcf_ID_low> AND <vcf_ID_high> AND POS >=
<input_high> LIMIT 1;
```

<vcf_ID_low> and <vcf_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 P OS 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.

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.

Figure 18. Sample 16-character ID Query, Part 1.

Figure 19. Sample 16-character ID Query, Part 2.

Figure 20. Sample 16-character ID Query, Part 3.

Figure 21. Sample 16-character ID Query, Part 4.

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 N/A character in the query result would be hidden as expected (**Figure 22**).

Figure 22. Sample 13-character ID Query, Part 1.

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

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 one-taping copying from the resulting VCF table to Excel spreadsheets is supported.

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

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**.

Alfalfa Database

Gene ID:

Annotation Keywords:

Seqid: POS from: to   (Range <= 3000bp)

Input gene ID was invalid.

Figure 27. Sample ID Query with invalid ID input.

Alfalfa Database

Gene ID:

Annotation Keywords:

Seqid: POS from: to   (Range <= 3000bp)

No such keywords found in annotation records.

Figure 28. Sample Keyword Query with no matched result.

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@wtlab alfalfa_database_bak]# tree -h ./static/
./static/
|-- [ 30]  css
|   |-- [ 734]  main.css
|-- [ 170]  DNA
|   |-- [ 98M]  Chr1.txt
|   |-- [ 81M]  Chr2.txt
|   |-- [ 99M]  Chr3.txt
|   |-- [ 88M]  Chr4.txt
|   |-- [105M]  Chr5.txt
|   |-- [109M]  Chr6.txt
|   |-- [ 91M]  Chr7.txt
|   |-- [ 87M]  Chr8.txt
|-- [ 144]  sql
|   |-- [7.0M]  Alfalfa_Database.jar
|   |-- [114G]  Alfalfa.db
|   |-- [ 28]  get_max_vcfID.sql
|   |-- [ 456]  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 generate 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.


```
> seqdata
BStringSet object of length 657:
      width seq                      names
[1] 102287719 TTTACTCGTTTTCAACTG...TATAGCATAAAGATGCAA Chr1
[2] 84774831  TGCCTTCAAGAGGTAAA...ATAACCCGATAGGTACC Chr2
[3] 104248068  AATCAAATCTCACTTCTT...GAAAGACCCGGGTGGAGT Chr3
[4] 92728752  CTAAACCCCTAAACCCGAA...CATACTCCATGATGTTAT Chr4
[5] 109743485  AATTTGTGGAGGAATCAA...CAACACGTTTGCGAACTC Chr5
...
[653] 23751  ACACTTCGGATGACAGAA...AAGTCTAAATCAAATCA contig645end
[654] 21271  TGTATTAATTTGTGTAAA...ATCGGGAAGGCTAAGGAG contig646end
[655] 25249  TCCCCACCTCGAAATTT...TGTGATTAACCAAGTTGC contig647end
[656] 12176  ACATGATTATGGTCAAAG...ATCGAATTCACCGATATG contig648end
[657] 21429  GCAATACCAATGGTGTT...TATAATTTGAAGGTAGAA contig649end
```

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

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.

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

```

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.     "GeneID" 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,

```

```

26. "leaf_bud_rep1" REAL NOT NULL,
27. "leaf_bud_rep2" REAL NOT NULL,
28. "leaf_bud_rep3" REAL NOT NULL,
29. "main_root_rep1" REAL NOT NULL,
30. "mature_leaf_rep1" REAL NOT NULL,
31. "mature_leaf_rep2" REAL NOT NULL,
32. "mature_leaf_rep3" REAL NOT NULL,
33. "MS_trueleaf_rep1" REAL NOT NULL,
34. "MS_xia_rep1" REAL NOT NULL,
35. "MS_ziye_rep1" REAL NOT NULL,
36. "nod_21_rep1" REAL NOT NULL,
37. "nod_28_rep1" REAL NOT NULL,
38. "seed_rep1" REAL NOT NULL,
39. "stem_branch_rep1" REAL NOT NULL,
40. "stem_rep1" REAL NOT NULL,
41. "yige_rep1" REAL NOT NULL,
42. "yige_rep2" REAL NOT NULL,
43. "yige_rep3" REAL NOT NULL,
44. "young_leaf_rep1" REAL NOT NULL,
45. "young_leaf_rep2" REAL NOT NULL,
46. "young_leaf_rep3" REAL NOT NULL,
47. PRIMARY KEY("GeneID")
48. );
49.
50. CREATE TABLE "annotate_info" (
51. "ID_full" TEXT NOT NULL,
52. "db_name" TEXT NOT NULL,
53. "info1" TEXT,
54. "info2" TEXT,
55. CONSTRAINT "FK_annotate" FOREIGN KEY("ID_full") REFERENCES "pros_info"("ID_full")
56. );
57.
58. CREATE TABLE "gff" (
59. "unique_id" INTEGER NOT NULL UNIQUE,
60. "seqid" TEXT NOT NULL,
61. "source" TEXT NOT NULL,
62. "type" TEXT NOT NULL,
63. "start" INTEGER NOT NULL,
64. "end" INTEGER NOT NULL,
65. "score" TEXT NOT NULL,
66. "strand" TEXT NOT NULL,
67. "phase" TEXT NOT NULL,
68. "attributes" TEXT NOT NULL,
69. "DNA" TEXT NOT NULL,
70. PRIMARY KEY("unique_id" AUTOINCREMENT)
71. );
72.
73. CREATE TABLE "go_info" (
74. "ID_full" TEXT NOT NULL UNIQUE,
75. "go" TEXT NOT NULL,
76. PRIMARY KEY("ID_full"),
77. CONSTRAINT "FK_go" FOREIGN KEY("ID_full") REFERENCES "pros_info"("ID_full")
78. );
79.
80. CREATE TABLE "interpro_info" (
81. "ID_full" TEXT NOT NULL UNIQUE,
82. "interpro" TEXT NOT NULL,
83. PRIMARY KEY("ID_full"),
84. CONSTRAINT "FK_interpro" FOREIGN KEY("ID_full") REFERENCES "pros_info"("ID_full")
85. );
86.

```

```

87. CREATE TABLE "kegg_info" (
88.     "ID_full" TEXT NOT NULL UNIQUE,
89.     "kegg" TEXT NOT NULL,
90.     PRIMARY KEY("ID_full"),
91.     CONSTRAINT "FK_kegg" FOREIGN KEY("ID_full") REFERENCES "pros_info"("ID_full")
92. );
93.
94. CREATE TABLE "pathway_info" (
95.     "ID_full" TEXT NOT NULL UNIQUE,
96.     "pathway" TEXT NOT NULL,
97.     PRIMARY KEY("ID_full"),
98.     CONSTRAINT "FK_pathway" FOREIGN KEY("ID_full") REFERENCES "pros_info"("ID_full")
99. );
100.
101. CREATE TABLE "pfam_info" (
102.     "ID_full" TEXT NOT NULL UNIQUE,
103.     "pfam" TEXT NOT NULL,
104.     PRIMARY KEY("ID_full"),
105.     CONSTRAINT "FK_pfam" FOREIGN KEY("ID_full") REFERENCES "pros_info"("ID_full")
106. );
107.
108. CREATE TABLE "pros_info" (
109.     "unique_id" INTEGER NOT NULL UNIQUE,
110.     "ID_full" TEXT NOT NULL UNIQUE,
111.     "pros" TEXT NOT NULL,
112.     CONSTRAINT "FK" FOREIGN KEY("unique_id") REFERENCES "gff"("unique_id"),
113.     PRIMARY KEY("unique_id","ID_full")
114. );
115.
116. CREATE TABLE sqlite_sequence(name,seq);
117.
118. CREATE TABLE "vcf" (
119.     "vcf_ID" INTEGER NOT NULL UNIQUE,
120.     "CHROM" TEXT NOT NULL,
121.     "POS" INTEGER NOT NULL,
122.     "REF" TEXT NOT NULL,
123.     "ALT" TEXT NOT NULL,
124.     "C1" TEXT NOT NULL,
125.     "C10" TEXT NOT NULL,
126.     "C100" TEXT NOT NULL,
127.     "C101" TEXT NOT NULL,
128.     "C102" TEXT NOT NULL,
129.     "C103" TEXT NOT NULL,
130.     "C104" TEXT NOT NULL,
131.     "C105" TEXT NOT NULL,
132.     "C106" TEXT NOT NULL,
133.     "C107" TEXT NOT NULL,
134.     "C108" TEXT NOT NULL,
135.     "C109" TEXT NOT NULL,
136.     "C11" TEXT NOT NULL,
137.     "C110" TEXT NOT NULL,
138.     "C111" TEXT NOT NULL,
139.     "C112" TEXT NOT NULL,
140.     "C113" TEXT NOT NULL,
141.     "C114" TEXT NOT NULL,
142.     "C115" TEXT NOT NULL,
143.     "C116" TEXT NOT NULL,
144.     "C117" TEXT NOT NULL,
145.     "C118" TEXT NOT NULL,
146.     "C119" TEXT NOT NULL,
147.     "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,	

209. "C52" TEXT NOT NULL,
 210. "C53" TEXT NOT NULL,
 211. "C54" TEXT NOT NULL,
 212. "C55" TEXT NOT NULL,
 213. "C56" TEXT NOT NULL,
 214. "C57" TEXT NOT NULL,
 215. "C58" TEXT NOT NULL,
 216. "C59" TEXT NOT NULL,
 217. "C6" TEXT NOT NULL,
 218. "C60" TEXT NOT NULL,
 219. "C61" TEXT NOT NULL,
 220. "C62" TEXT NOT NULL,
 221. "C63" TEXT NOT NULL,
 222. "C64" TEXT NOT NULL,
 223. "C65" TEXT NOT NULL,
 224. "C66" TEXT NOT NULL,
 225. "C67" TEXT NOT NULL,
 226. "C68" TEXT NOT NULL,
 227. "C69" TEXT NOT NULL,
 228. "C7" TEXT NOT NULL,
 229. "C70" TEXT NOT NULL,
 230. "C71" TEXT NOT NULL,
 231. "C72" TEXT NOT NULL,
 232. "C73" TEXT NOT NULL,
 233. "C74" TEXT NOT NULL,
 234. "C75" TEXT NOT NULL,
 235. "C76" TEXT NOT NULL,
 236. "C77" TEXT NOT NULL,
 237. "C78" TEXT NOT NULL,
 238. "C79" TEXT NOT NULL,
 239. "C8" TEXT NOT NULL,
 240. "C80" TEXT NOT NULL,
 241. "C81" TEXT NOT NULL,
 242. "C82" TEXT NOT NULL,
 243. "C83" TEXT NOT NULL,
 244. "C84" TEXT NOT NULL,
 245. "C85" TEXT NOT NULL,
 246. "C86" TEXT NOT NULL,
 247. "C87" TEXT NOT NULL,
 248. "C88" TEXT NOT NULL,
 249. "C89" TEXT NOT NULL,
 250. "C9" TEXT NOT NULL,
 251. "C90" TEXT NOT NULL,
 252. "C91" TEXT NOT NULL,
 253. "C92" TEXT NOT NULL,
 254. "C93" TEXT NOT NULL,
 255. "C94" TEXT NOT NULL,
 256. "C95" TEXT NOT NULL,
 257. "C96" TEXT NOT NULL,
 258. "C97" TEXT NOT NULL,
 259. "C98" TEXT NOT NULL,
 260. "C99" TEXT NOT NULL,
 261. "L1" TEXT NOT NULL,
 262. "L10" TEXT NOT NULL,
 263. "L11" TEXT NOT NULL,
 264. "L12" TEXT NOT NULL,
 265. "L13" TEXT NOT NULL,
 266. "L15" TEXT NOT NULL,
 267. "L16" TEXT NOT NULL,
 268. "L17" TEXT NOT NULL,
 269. "L18" TEXT NOT NULL,

```

270. "L19" TEXT NOT NULL,
271. "L2" TEXT NOT NULL,
272. "L20" TEXT NOT NULL,
273. "L21" TEXT NOT NULL,
274. "L22" TEXT NOT NULL,
275. "L27" TEXT NOT NULL,
276. "L3" TEXT NOT NULL,
277. "L32" TEXT NOT NULL,
278. "L33" TEXT NOT NULL,
279. "L4" TEXT NOT NULL,
280. "L5" TEXT NOT NULL,
281. "L6" TEXT NOT NULL,
282. "L7" TEXT NOT NULL,
283. "L8" TEXT NOT NULL,
284. "L9" TEXT NOT NULL,
285. "MsREF" TEXT NOT NULL,
286. "zhonglan" TEXT NOT NULL,
287. PRIMARY KEY("vcf_ID" AUTOINCREMENT)
288. );

```

10.2 vcf.java

```

1. import java.io.BufferedReader;
2. import java.io.FileInputStream;
3. import java.io.IOException;
4. import java.io.InputStreamReader;
5. import java.util.ArrayList;
6.
7. public class vcf {
8.
9.     public static void main(String[] args) throws IOException {
10.         String vcf = args[0]; // Usage: java vcf <path to vcf>
11.         FileInputStream fis = new FileInputStream(vcf);
12.         BufferedReader br = new BufferedReader(new InputStreamReader(fis));
13.         String line = null;
14.         int cnt = 1; // vcf ID
15.         while ((line = br.readLine()) != null) {
16.             // ignore file headers and column names
17.             if (line.length() == 0 || line.charAt(0) == '#')
18.                 continue;
19.             String[] strArrays = line.split("\s+");
20.             StringBuilder fullLine = new StringBuilder();
21.             int i = 9;
22.
23.             // Read variation info
24.             fullLine.append(strArrays[0]); // CHROM
25.             fullLine.append("\t");
26.             fullLine.append(strArrays[1]); // POS
27.             fullLine.append("\t");
28.             fullLine.append(strArrays[3]); // REF
29.             fullLine.append("\t");
30.             fullLine.append(strArrays[4]); // ALT
31.             fullLine.append("\t");
32.
33.             // Split REF into lists
34.             ArrayList<String> ref = new ArrayList<String>();
35.             ref.add(strArrays[3]);
36.             String[] s2 = strArrays[4].split(",");
37.             for (String s1 : s2) {
38.                 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,C14,C15,C16,C17,C18,C19,C2,C20,C21,C22,C23,C24,C25,C26,C27,C28,C29,"
63.         + "C3,C30,C31,C32,C33,C34,C35,C36,C37,C38,C39,C4,C40,C41,C42,C43,C44,C45,C46,C47,"
64.         + "C48,C49,C5,C50,C51,C52,C53,C54,C55,C56,C57,C58,C59,C6,C60,C61,C62,C63,C64,C65,"
65.         + "C66,C67,C68,C69,C7,C70,C71,C72,C73,C74,C75,C76,C77,C78,C79,C8,C80,C81,C82,C83,"
66.         + "C84,C85,C86,C87,C88,C89,C9,C90,C91,C92,C93,C94,C95,C96,C97,C98,C99,L1,L10,L11,"
67.         + "L12,L13,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 += "\"" + 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':

```

```

16.         ans += 'G'
17.     elif rev[i] == 'G':
18.         ans += 'C'
19.     return ans
20.
21. # home page with input place holders
22. @app.route('/', methods = ['POST', 'GET'])
23. def home():
24.     return render_template('home.html', show_place_holder=True)
25.
26. # query mode A, ID search
27. @app.route('/ID/', methods = ['POST', 'GET'])
28. def mode_A():
29.     if request.method == 'POST':
30.         id = request.form['content1'] # gene ID input from html
31.
32.         if len(id) != 13 and len(id) != 16 and len(id) != 20:
33.             return render_template('error.html', err_msg="Input gene ID was invalid.")
34.
35.         conn = sqlite3.connect("./static/sql/Alfalfa.db")
36.         cu = conn.cursor()
37.
38.         # query pros info
39.         sql = "SELECT * FROM pros_info WHERE ID_full LIKE '\"' + id + \"%\\",\"
40.         cu.execute(sql)
41.         pros_infos = cu.fetchall()
42.
43.         if len(pros_infos) == 0:
44.             return render_template('error.html', err_msg="Gene ID not found")
45.
46.         # query annotate info
47.         sql = "SELECT * FROM annotate_info WHERE ID_full LIKE '\"' + id + \"%\\",\"
48.         cu.execute(sql)
49.         annotate_infos = cu.fetchall()
50.
51.         show_annotate = True
52.         if len(annotate_infos) == 0:
53.             show_annotate = False
54.
55.         # query go_info, interpre_info, kegg_info, pathway_info, and pfam_info
56.         have_info = False
57.
58.         pri_id = id
59.         if len(id) != 13:
60.             pri_id = id[0:13]
61.
62.         sql = "SELECT pros_info.ID_full, pros_info.unique_id, kegg_info.kegg, pathway_info.pathway,\"
63.         sql += " go_info.go, interpre_info.interpro, pfam_info.pfam FROM pros_info\\n\"
64.         sql += " LEFT OUTER JOIN go_info ON pros_info.ID_full = go_info.ID_full\\n\"
65.         sql += " LEFT OUTER JOIN interpre_info ON pros_info.ID_full = interpre_info.ID_full\\n\"
66.         sql += " LEFT OUTER JOIN kegg_info ON pros_info.ID_full = kegg_info.ID_full\\n\"
67.         sql += " LEFT OUTER JOIN pathway_info ON pros_info.ID_full = pathway_info.ID_full\\n\"
68.         sql += " LEFT OUTER JOIN pfam_info ON pros_info.ID_full = pfam_info.ID_full\\n\"
69.         sql += " WHERE pros_info.ID_full LIKE '\"' + pri_id + \"%\\",\"
70.         cu.execute(sql)
71.         infos = cu.fetchall()
72.
73.         # filter infos, only keep rows with at least one annotation that is not None
74.         info = []
75.         for x in infos:
76.             for i in range(2, 7):

```

```

77.         if x[i] != None:
78.             info = x
79.             break
80.
81.     # process kegg
82.     kegg_list = None
83.     if len(info) > 0:
84.         if info[2] != None:
85.             kegg_list = str(info[2]).split(";")
86.         if kegg_list != None:
87.             for i in range(0, len(kegg_list)):
88.                 kegg_list[i] = kegg_list[i][6:]
89.                 have_info = True
90.
91.     # process pathway
92.     temp_list = None
93.     pathway_list = []
94.     if len(info) > 0:
95.         if info[3] != None:
96.             temp_list = str(info[3]).split(";")
97.         if temp_list != None:
98.             for i in range(0, len(temp_list)):
99.                 # exclude KEGG duplicates
100.                if not ("KEGG" in temp_list[i][0:4]):
101.                    pathway_list.append(temp_list[i])
102.            if len(pathway_list) == 0:
103.                pathway_list = None
104.            else:
105.                have_info = True
106.
107.     # process go
108.     go_list = None
109.     if len(info) > 0:
110.         if info[4] != None:
111.             go_list = str(info[4]).split(";")
112.         if go_list != None:
113.             for i in range(0, len(go_list)):
114.                 go_list[i] = go_list[i][3:]
115.                 have_info = True
116.
117.     # process interpro
118.     interpro_list = None
119.     if len(info) > 0:
120.         if info[5] != None:
121.             interpro_list = str(info[5]).split("|")
122.         if interpro_list != None:
123.             have_info = True
124.
125.     # process pfam
126.     pfam_list = None
127.     if len(info) > 0:
128.         if info[6] != None:
129.             pfam_list = str(info[6]).split(";")
130.         if pfam_list != None:
131.             have_info = True
132.
133.     # get 16-character gene ID
134.     gene_ID = None
135.     if have_info:
136.         gene_ID = str(info[0])[0:16]
137.

```



```

138.     # query gff
139.     sql = "SELECT * FROM gff AS G, ID_to_uniqueid AS I WHERE I.ID = \"\" + pri_id + \"\" AND G.unique_id >= I.star
t AND G.unique_id <= I.end;"
140.     cu.execute(sql)
141.     temp_list = cu.fetchall()
142.     gffs = []
143.     for x in temp_list:
144.         if id in x[9]:
145.             gffs.append(x)
146.
147.     # replace by DNA conjugates when needed
148.     for i in range(0, len(gffs)):
149.         if gffs[i][7] == "-":
150.             temp = list(gffs[i])
151.             temp[10] = get_conjugate(temp[10])
152.             gffs[i] = tuple(temp)
153.
154.     # add combined CDS sequences to pros_info
155.     cds = []
156.     for x in gffs:
157.         if x[3] == "CDS":
158.             cds.append(x)
159.     for i in range(0, len(pros_infos)):
160.         cds_combined = ""
161.         for x in cds:
162.             if str(pros_infos[0][1]) in str(x[9]):
163.                 cds_combined += x[10]
164.             temp_list = list(pros_infos[i])
165.             temp_list.append(cds_combined)
166.             pros_infos[i] = tuple(temp_list)
167.
168.     # get transcriptome data from Table alltpm
169.     sql = "SELECT * FROM alltpm WHERE GeneID = \"\" + pri_id + ".01\";"
170.     cu.execute(sql)
171.     temp_list = cu.fetchall()[0]
172.     transcriptome_list = []
173.     for i in range(1, len(temp_list)):
174.         transcriptome_list.append(temp_list[i])
175.
176.     # print to html page
177.     if show_annotate:
178.         if have_info:
179.             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_i
nfos, gffs=gffs, len_gff=len(gffs), transcriptome_list=transcriptome_list)
180.         else:
181.             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)
182.         else:
183.             if have_info:
184.                 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=pf
am_list, pros_infos=pros_infos, gffs=gffs, len_gff=len(gffs), transcriptome_list=transcriptome_list)
185.             else:
186.                 return render_template("home.html", mode="A", id=id, show_annotate=show_annotate, have_info=have_info, pr
os_infos=pros_infos, gffs=gffs, len_gff=len(gffs), transcriptome_list=transcriptome_list)
187.             return redirect("/")
188.
189.     # query mode B, keywords search

```

```

190. @app.route("/annotation", methods = ['POST', 'GET'])
191. def mode_B():
192.     if request.method == 'POST':
193.         keywords = request.form['content2'] # keywords input from html
194.
195.         if len(keywords) < 1:
196.             return render_template('error.html', err_msg="No keywords entered.")
197.
198.         # query annotate_info
199.         conn = sqlite3.connect("./static/sql/Alfalfa.db")
200.         cu = conn.cursor()
201.         sql = "SELECT * FROM annotate_info WHERE info1 LIKE '%" + keywords + "%' OR info2 LIKE '%" + keywords + "%'";
202.         cu.execute(sql)
203.         ans_list = cu.fetchall()
204.
205.         if len(ans_list) == 0:
206.             return render_template('error.html', err_msg="No such keywords found in annotation records.")
207.
208.         # combine rows with the same gene ID by calculating rowspan value for html
209.         rowspan = []
210.         temp = ans_list[0][0]
211.         cnt = 0
212.         l = 0
213.         for ans in ans_list:
214.             if ans[0] == temp:
215.                 cnt += 1
216.             else:
217.                 if len(rowspan) == 0:
218.                     rowspan.append(cnt)
219.                     for i in range(1, cnt):
220.                         rowspan.append(0)
221.                 else:
222.                     l += 1
223.                     rowspan.append(cnt + 1)
224.                     for i in range(1, cnt + 1):
225.                         rowspan.append(0)
226.                 temp = ans[0]
227.                 cnt = 0
228.             rowspan.append(len(ans_list) - len(rowspan))
229.         for i in range(0, len(ans_list) - len(rowspan)):
230.             rowspan.append(0)
231.
232.         # add rowspan information to each row of query results
233.         for i in range(0, len(ans_list)):
234.             temp_list = []
235.             temp_list.append(rowspan[i])
236.             ans_list[i] += tuple(temp_list)
237.
238.         # print to html page
239.         for i in range(0, len(keywords)):
240.             if keywords[i] == " ":
241.                 keywords[i] = "~"
242.         return render_template('home.html', mode="B", keywords=keywords, ans_list=ans_list, l=l + 2)
243.     return redirect("/")
244.
245. # query mode C, VCF search
246. @app.route("/VCF", methods = ['POST', 'GET'])
247. def mode_C():
248.     if request.method == 'POST':
249.         seqid = request.form['content3'] # seqid input from html

```

```

250.     low = request.form['content4'] # low input from html
251.     high = request.form['content5'] # high input from html
252.
253.     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):
254.         return render_template('error.html', err_msg="Invalid VCF query.")
255.
256.     # default range is set to be 1 ~ 3000 bp
257.     if int(high) - int(low) >= 3000:
258.         return render_template('error.html', err_msg="Please select a range that is less than or equals to 3000bp.")
259.
260.     conn = sqlite3.connect("./static/sql/Alfalfa.db")
261.     cu = conn.cursor()
262.
263.     # setup ranges from sql results
264.     max_pos = 148124263 # get max vcflID.sql
265.     arr = [1, 17551074, 32991065, 52346699, 69388115, 89626087, 113452234, 132015073, max_pos + 1] # get_min_vc
        flDs.sql
266.
267.     # get a smaller vcf ID range to query from
268.     temp_low = str(arr[int(seqid[3:])] - 1)
269.     temp_high = str(arr[int(seqid[3:])] - 1)
270.     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;"
271.     cu.execute(sql)
272.     temp = cu.fetchall()
273.     start = str(temp_low)
274.     if len(temp) != 0:
275.         start = str(list(temp[0])[0])
276.     sql = "SELECT vcf_ID FROM vcf_jump_table WHERE vcf_ID BETWEEN " + temp_low + " AND " + temp_high +
        " AND POS >= " + high + " LIMIT 1;"
277.     cu.execute(sql)
278.     temp = cu.fetchall()
279.     end = str(temp_high)
280.     if len(temp) != 0:
281.         end = str(list(temp[0])[0])
282.
283.     # query VCF records
284.     sql = "SELECT * FROM vcf WHERE vcf_ID BETWEEN " + start + " AND " + end + " AND POS BETWEEN " + lo
        w + " AND " + high + ";"
285.     cu.execute(sql)
286.     vcfs = cu.fetchall()
287.
288.     if len(vcfs) == 0:
289.         return render_template('error.html', err_msg="No VCF entry found.")
290.
291.     # get DNA sequence between low and high
292.     file_name = "./static/DNA/" + seqid + ".txt"
293.     f = open(file_name, 'r')
294.     data = f.read()
295.     f.close()
296.     seq = data[int(low) - 1:int(high)]
297.
298.     # print to html page
299.     return render_template('home.html', mode="C", seqid=int(seqid[3:]), low=low, high=high, length=len(seq), DNA=se
        q, vcfs=vcfs, l=len(vcfs))
300.     return redirect("/")
301.
302. if __name__ == "__main__":
303.     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 %} {% endblock %}
9. </head>
10. <body>
11.   {% block body %} {% endblock %}
12. </body>
13. </html>

```

10.5 home.html

```

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

```

```

42.         (Range <= 3000bp)
43.     </form>
44. <p></p>
45. <form action="/" method = 'POST'>
46.     <input type="submit" value="Reset">
47. </form>
48. </div>
49. <p></p>
50. <div style="text-align: left; width: 80%;">
51.     <h2>Users' Guide:</h2>
52.     <h3>1. An valid Gene ID could have 13 (MsG1A10000001), 16 (MsG1A10000001.01), or 20 (MsG1A10000001.01.
53. T01) characters.</h3>
54.     <h3>2. When querying about a gene ID, all related genome sequencing, transcriptome sequencing, and annotation dat
55. a would be shown.</h3>
56.     <h3>3. When querying about a keyword, the gene IDs of all annotation records containing that keyword would be fou
57. nd (case insensitive).</h3>
58.     <h3>4. To improve page loading time, the range of VCF queries was restricted to be 1 ~ 3000 basepairs.</h3>
59. </div>
60. <form action="/ID/" method = 'POST'>
61.     Gene ID:
62.     <input type="text" name="content1" id="content1" value={{ id }}>
63.     <input type="submit" value="Search gene ID">
64. </form>
65. <p></p>
66. <form action="/annotation/" method = 'POST'>
67.     Annotation Keywords:
68.     <!-- handle spaces within the given keywords -->
69.     <script>
70.         var jsVar = '{{ keywords }}';
71.         var echo = "";
72.         for (var i = 0; i < jsVar.length; i++) {
73.             if(jsVar.charAt(i) == '~')
74.                 echo += ' ';
75.             else
76.                 echo += jsVar.charAt(i);
77.         }
78.         document.write("<input type='text' name='content2' id='content2' value='" + echo + "'>");
79.     </script>
80.     <input type="submit" value="Search annotation">
81. </form>
82. <p></p>
83. <form action="/VCF/" method = 'POST'>
84.     Seqid:
85.     <select name="content3" id="content3">
86.         <script type="text/javascript">
87.             var num = '{{ seqid }}';
88.             for(var i = 1; i < 9; i++) {
89.                 if(i.toString() == num)
90.                     document.write("<option value='Chr" + i + "' selected>Chr" + i + "</option>");
91.                 else
92.                     document.write("<option value='Chr" + i + "'>Chr" + i + "</option>");
93.             }
94.         </script>
95.     </select>
96.     POS from:
97.     <input type="text" name="content4" id="content4" value={{ low }}>
98.     to
99.     <input type="text" name="content5" id="content5" value={{ high }}>

```

```

100.     <input type="submit" value="Search VCF">
101.     (Range <= 3000bp)
102. </form>
103. <p></p>
104. <form action="/" method = 'POST'>
105.     <input type="submit" value="Reset">
106. </form>
107. </div>
108. {% endif %}
109.
110. <!-- query mode A, ID search -->
111. {% if mode == "A" %}
112.     <div><h2>Query Results:</h2></div>
113.     <div class="container">
114.         <!-- each 16-character gene ID should have at most one set of these info -->
115.         {% if have_info %}
116.             <table>
117.                 <thead><th>Annotation Links</th></thead>
118.                 <tbody><tr><td>
119.                     <div class="container" style="text-align: left;">
120.                         <h3>Gene ID: {{ gene_ID }}</h3>
121.
122.                         {% if kegg_list != None %}
123.                         <h3>KEGG:
124.                             {% for kegg in kegg_list %}
125.                             <a href="https://www.kegg.jp/entry/K{{ kegg }}" target="_blank">{{ kegg }}</a>;
126.                             {% endfor %}
127.                         </h3>
128.                         {% endif %}
129.
130.                         {% if pathway_list != None %}
131.                         <h3>Pathway:
132.                             {% for pathway in pathway_list %}
133.                             <a style="text-decoration: underline;">{{ pathway }}</a>;
134.                             {% endfor %}
135.                         </h3>
136.                         {% endif %}
137.
138.                         {% if go_list != None %}
139.                         <h3>GO:
140.                             {% for go in go_list %}
141.                             <a href="http://amigo.geneontology.org/amigo/term/GO:{{ go }}" target="_blank">{{ go }}</a>;
142.                             {% endfor %}
143.                         </h3>
144.                         {% endif %}
145.
146.                         {% if interpro_list != None %}
147.                         <h3>Interpro:
148.                             {% for interpro in interpro_list %}
149.                             <script type="text/javascript">
150.                                 var jsVar = '{{ interpro }}';
151.                                 var l = jsVar.length;
152.                                 var description = jsVar.substr(0, l - 11)
153.                                 var code = jsVar.substr(l - 10, 9)
154.                                 document.write(description + "(<a href='https://www.ebi.ac.uk/interpro/entry/InterPro/' + code +
155.                                     '/' target='_blank'>" + code + "</a>);")
156.                             </script>
157.                             {% endfor %}
158.                         </h3>
159.                         {% endif %}

```



```

160.         {% if pfam_list != None %}
161.         <h3>Pfam:
162.         {% for pfam in pfam_list %}
163.         <a href="https://pfam.xfam.org/family/{{ pfam }}" target="_blank">{{ pfam }}</a>;
164.         {% endfor %}
165.         </h3>
166.         {% endif %}
167.     </div>
168. </td></tr></tbody>
169. </table>
170. <p></p>
171. {% endif %}
172.
173. <!-- each gene ID query should have at most one set of annotations -->
174. {% if show_annotate %}
175.     <table>
176.     <thead>
177.     <th>Gene ID</th>
178.     <th>db Name</th>
179.     <th>Info 1</th>
180.     <th>Info 2</th>
181.     </thead>
182.     <tbody>
183.     <!-- combine dable cells of gene ID -->
184.     {% if annotate_span != 0 %}
185.     <td rowspan="{{ annotate_span }}">{{ temp[0] }}</td>
186.     {% endif %}
187.     {% for annotate_info in annotate_infos %}
188.     <tr>
189.     <td>{{ annotate_info[1] }}</td>
190.     <td>{{ annotate_info[2] }}</td>
191.     <td>{{ annotate_info[3] }}</td>
192.     </tr>
193.     {% endfor %}
194.     </tbody>
195.     </table>
196.     <p></p>
197. {% endif %}
198.
199. <!-- visualize transcriptome data using echarts -->
200. <div id="main" style="border: 2px solid #aaa; width: 98%; height:500px; background-
color: rgb(212, 239, 248); overflow-x: scroll;"></div>
201. <script type="text/javascript">
202.     // convert Python list to js array
203.     var jsVar = '{{ transcriptome_list }}'.toString();
204.     var jsArray = new Array();
205.     jsVarjsVar = jsVar.substr(1, jsVar.length - 2);
206.     while(jsVar.length > 0) {
207.         var index = jsVar.indexOf(',');
208.         if(index == -1)
209.             break;
210.         var temp = jsVar.substr(0, index - 1);
211.         jsArray.push(Number(temp));
212.         jsVarjsVar = jsVar.substr(index + 2);
213.     }
214.     jsArray.push(Number(jsVar));
215.
216.     // render the echarts
217.     var myChart = echarts.init(document.getElementById('main'));
218.     var option = {
219.         title: {

```

```

220.         text: 'Expression in different Tissues'
221.     },
222.     tooltip: {},
223.     xAxis: {
224.         axisLabel: {
225.             show: true,
226.             interval: 0,
227.             rotate: 45
228.         },
229.         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_rep
1","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_le
af_rep2","young_leaf_rep3"]
230.     },
231.     yAxis: {},
232.     grid: {
233.         y2: 80
234.     },
235.     series: [{
236.         name: 'TPM',
237.         type: 'bar',
238.         data: jsArray
239.     }]
240. };
241. myChart.setOption(option);
242. </script>
243. <p></p>
244.
245. <!-- each valid 20-character gene ID should have one set of pros_info with complete CDS seq -->
246. <table>
247.     {% for pros_info in pros_infos %}
248.     <thead><th>Gene ID: {{ pros_info[1] }}</th></thead>
249.     <tbody>
250.         <tr><td style="background-color: bisque;">Protein Sequence</td></tr>
251.         <tr><td class="seq">{{ pros_info[2] }}</td></tr>
252.         <tr><td style="background-color: bisque;">Complete CDS Sequence</td></tr>
253.         <tr><td class="seq">{{ pros_info[3] }}</td></tr>
254.     </tbody>
255.     {% endfor %}
256. </table>
257.
258. <!-- each valid 20-character gene ID should have some gff records -->
259. <div><h3>{{ len_gff }} gff entries were found:</h3></div>
260. <table>
261.     <thead>
262.         <th>Seqid</th>
263.         <th>Source</th>
264.         <th>Type</th>
265.         <th>Start</th>
266.         <th>End</th>
267.         <th>Score</th>
268.         <th>Strand</th>
269.         <th>Phase</th>
270.         <th>Attributes</th>
271.     </thead>
272.     <tbody>
273.         {% for gff in gffs %}
274.         <tr style="background-color: bisque;">
275.             <td>{{ gff[1] }}</td>

```

```

276.         <td>{{ gff[2] }}</td>
277.         <td>{{ gff[3] }}</td>
278.         <td>{{ gff[4] }}</td>
279.         <td>{{ gff[5] }}</td>
280.         <td>{{ gff[6] }}</td>
281.         <td>{{ gff[7] }}</td>
282.         <td>{{ gff[8] }}</td>
283.         <td style="overflow-x: scroll;">{{ gff[9] }}</td>
284.     </tr>
285.     <tr><td colspan="10" class="seq">{{ gff[10] }}</td></tr>
286.     {% endfor %}
287. </tbody>
288. </table>
289. </div>
290. <p></p>
291. <div><h3>Note: Double-
    clicking on any of the sequences could select the entire sequence for copying purposes.</h3></div>
292. <p></p>
293. {% endif %}
294.
295. <!-- query mode B, keywords search -->
296. {% if mode == "B" %}
297.     <div><h2>{{ l }} gene IDs were found:</h2></div>
298.     <div class="container">
299.         <table>
300.             <thead>
301.                 <th>Gene ID</th>
302.                 <th>db Name</th>
303.                 <th>Info 1</th>
304.                 <th>Info 2</th>
305.             </thead>
306.             <tbody>
307.                 {% for ans in ans_list %}
308.                     <tr>
309.                         <!-- combine data cells and add hyper link for each 20-character gene ID found -->
310.                         <script type="text/javascript">
311.                             var jsVar = '{{ ans[4] }}';
312.                             var hyper = "<form action='/ID/' method = 'POST'><input type='text' name='content1' id='content1' value
313.                             ={{ ans[0] }} hidden><input type='submit' value='?'>";
314.                             if(jsVar != "0")
315.                                 document.write("<td rowspan=" + jsVar + ">" + hyper + "{{ ans[0] }}</form></td>");
316.                         </script>
317.                         <td>{{ ans[1] }}</td>
318.                         <td>{{ ans[2] }}</td>
319.                         <td>{{ ans[3] }}</td>
320.                     </tr>
321.                 {% endfor %}
322.             </tbody>
323.         </table>
324.     </div>
325.     <p></p>
326.     <div><h3>Note: Clicking on the question mark next to a gene ID would redirect to the query results about that gene ID.
327.     </h3></div>
328.     <p></p>
329. {% endif %}
330.
331. <!-- query mode C, VCF search -->
332. {% if mode == "C" %}
333.     <div><h2>Querying the following gene segment:</h2></div>
334.     <!-- show information about the selected DNA segment -->
335.     <div>

```

```

334.     <table style="width: 80%;">
335.         <thead>
336.             <tr>
337.                 <th>Seqid</th>
338.                 <th>Start</th>
339.                 <th>End</th>
340.                 <th>Total Length</th>
341.             </tr>
342.         </thead>
343.         <tbody>
344.             <tr style="background-color: bisque;">
345.                 <td>Chr{{ seqid }}</td>
346.                 <td>{{ low }}</td>
347.                 <td>{{ high }}</td>
348.                 <td>{{ length }}bp</td>
349.             </tr>
350.         </tbody>
351.         <thead><tr><th colspan="4">Main DNA Sequence</th></tr></thead>
352.         <tbody><tr><td colspan="4" class="seq">{{ DNA }}</td></tr></tbody>
353.     </table>
354. </div>
355.
356. <!-- visulize VCF records within the query range -->
357. <div><h2>{{ 1 }} VCF entries were found:</h2></div>
358. <div class="container">
359.     <table style="user-select: all;"> <!-- for copying to Excel -->
360.         <!-- VCF headers -->
361.         <thead>
362.             <tr>
363.                 <th>POS</th>
364.                 <th>REF</th>
365.                 <th>ALT</th>
366.                 <th>C1</th>
367.                 <th>C10</th>
368.                 <th>C100</th>
369.                 <th>C101</th>
370.                 <th>C102</th>
371.                 <th>C103</th>
372.                 <th>C104</th>
373.                 <th>C105</th>
374.                 <th>C106</th>
375.                 <th>C107</th>
376.                 <th>C108</th>
377.                 <th>C109</th>
378.                 <th>C11</th>
379.                 <th>C110</th>
380.                 <th>C111</th>
381.                 <th>C112</th>
382.                 <th>C113</th>
383.                 <th>C114</th>
384.                 <th>C115</th>
385.                 <th>C116</th>
386.                 <th>C117</th>
387.                 <th>C118</th>
388.                 <th>C119</th>
389.                 <th>C12</th>
390.                 <th>C120</th>
391.                 <th>C121</th>
392.                 <th>C122</th>
393.                 <th>C123</th>
394.                 <th>C124</th>

```

395.	<th>C125</th>
396.	<th>C126</th>
397.	<th>C127</th>
398.	<th>C128</th>
399.	<th>C129</th>
400.	<th>C13</th>
401.	<th>C130</th>
402.	<th>C131</th>
403.	<th>C132</th>
404.	<th>C133</th>
405.	<th>C134</th>
406.	<th>C135</th>
407.	<th>C136</th>
408.	<th>C137</th>
409.	<th>C14</th>
410.	<th>C15</th>
411.	<th>C16</th>
412.	<th>C17</th>
413.	<th>C18</th>
414.	<th>C19</th>
415.	<th>C2</th>
416.	<th>C20</th>
417.	<th>C21</th>
418.	<th>C22</th>
419.	<th>C23</th>
420.	<th>C24</th>
421.	<th>C25</th>
422.	<th>C26</th>
423.	<th>C27</th>
424.	<th>C28</th>
425.	<th>C29</th>
426.	<th>C3</th>
427.	<th>C30</th>
428.	<th>C31</th>
429.	<th>C32</th>
430.	<th>C33</th>
431.	<th>C34</th>
432.	<th>C35</th>
433.	<th>C36</th>
434.	<th>C37</th>
435.	<th>C38</th>
436.	<th>C39</th>
437.	<th>C4</th>
438.	<th>C40</th>
439.	<th>C41</th>
440.	<th>C42</th>
441.	<th>C43</th>
442.	<th>C44</th>
443.	<th>C45</th>
444.	<th>C46</th>
445.	<th>C47</th>
446.	<th>C48</th>
447.	<th>C49</th>
448.	<th>C5</th>
449.	<th>C50</th>
450.	<th>C51</th>
451.	<th>C52</th>
452.	<th>C53</th>
453.	<th>C54</th>
454.	<th>C55</th>
455.	<th>C56</th>

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>


```

517.         <th>L27</th>
518.         <th>L3</th>
519.         <th>L32</th>
520.         <th>L33</th>
521.         <th>L4</th>
522.         <th>L5</th>
523.         <th>L6</th>
524.         <th>L7</th>
525.         <th>L8</th>
526.         <th>L9</th>
527.         <th>MsREF</th>
528.         <th>zhonglan</th>
529.     </tr>
530. </thead>
531. <tbody>
532.     {% for vcf in vcfs %}
533.     <tr>
534.         <td style="background-color: bisque;">{{ vcf[2] }}</td>
535.         <td style="background-color: rgb(166, 238, 95); overflow-x: scroll;">{{ vcf[3] }}</td>
536.         <td style="background-color: rgb(245, 104, 238); overflow-x: scroll;">{{ vcf[4] }}</td>
537.         {% for x in vcf[5:] %}
538.             <!-- color each table cell accordingly -->
539.             <script type="text/javascript">
540.                 var jsVar = '{{ x }}';
541.                 if(jsVar == ".")
542.                     document.write("<td>.</td>");
543.                 else {
544.                     var index = jsVar.indexOf("");
545.                     var REF = '{{ vcf[3] }}';
546.                     if(jsVar.substr(0, index) == REF) {
547.                         if(jsVar.substr(index + 1) == REF)
548.                             document.write("<td style='background-color: rgb(166, 238, 95); overflow-
x: scroll;'>" + jsVar + "</td>");
549.                         else
550.                             document.write("<td style='background-color: rgb(255, 172, 17); overflow-
x: scroll;'>" + jsVar + "</td>");
551.                     }
552.                     else {
553.                         if(jsVar.substr(index + 1) != REF) {
554.                             if(jsVar.substr(0, index) == jsVar.substr(index + 1))
555.                                 document.write("<td style='background-color: rgb(245, 104, 238); overflow-
x: scroll;'>" + jsVar + "</td>");
556.                             else
557.                                 document.write("<td style='background-color: yellow; overflow-
x: scroll;'>" + jsVar + "</td>");
558.                         }
559.                         else
560.                             document.write("<td style='background-color: rgb(255, 172, 17); overflow-
x: scroll;'>" + jsVar + "</td>");
561.                     }
562.                 }
563.             </script>
564.             {% endfor %}
565.         </tr>
566.     {% endfor %}
567. </tbody>
568. </table>
569. </div>
570. <p></p>
571. <div><h3>Legend:
572.     <span style="background-color: rgb(166, 238, 95);">REF gene type</span>;

```

```

573.     <span style="background-color: rgb(255, 172, 17);">Het with a REF</span>;
574.     <span style="background-color: yellow;">Het without a REF</span>;
575.     <span style="background-color: rgb(245, 104, 238);">Homo mutation</span>
576.   </h3></div>
577.   <div><h3>Note: Clicking on any part of the table above could select the entire table so that it could be copied to an Excel spreadsheet.</h3></div>
578.   <p></p>
579. {% endif %}
580.
581. {% endblock %}

```

10.6 error.html

```

1.   {% extends 'base.html' %}
2.
3.   {% block head %}
4.   <title>Alfalfa Database</title>
5.   {% endblock %}
6.
7.   {% block body %}
8.   <div><h1>Alfalfa Database</h1></div>
9.
10.  <div>
11.    <form action="/ID/" method = 'POST'>
12.      Gene ID:
13.      <input type="text" name="content1" id="content1" placeholder="MsG1A10000001">
14.      <input type="submit" value="Search gene ID">
15.    </form>
16.    <p></p>
17.    <form action="/annotation/" method = 'POST'>
18.      Annotation Keywords:
19.      <input type="text" name="content2" id="content2" placeholder="Heat shock factor">
20.      <input type="submit" value="Search annotation">
21.    </form>
22.    <p></p>
23.    <form action="/VCF/" method = 'POST'>
24.      Seqid:
25.      <select name="content3" id="content3">
26.        <option value="Chr1">Chr1</option>
27.        <option value="Chr2">Chr2</option>
28.        <option value="Chr3">Chr3</option>
29.        <option value="Chr4">Chr4</option>
30.        <option value="Chr5">Chr5</option>
31.        <option value="Chr6">Chr6</option>
32.        <option value="Chr7">Chr7</option>
33.        <option value="Chr8">Chr8</option>
34.      </select>
35.      POS from:
36.      <input type="text" name="content4" id="content4" placeholder="1">
37.      to
38.      <input type="text" name="content5" id="content5" placeholder="300">
39.      <input type="submit" value="Search VCF">
40.      (Range <= 3000bp)
41.    </form>
42.    <p></p>
43.    <form action="/" method = 'POST'>
44.      <input type="submit" value="Reset">
45.    </form>
46.  </div>
47.

```

```

48. <div><h2>{{ err_msg }}</h2></div>
49.
50. {% endblock %}

```

10.7 main.css

```

1.  body, html {
2.      font-family: Arial, Helvetica, sans-serif;
3.      background-color: rgb(228, 226, 226);
4.  }
5.
6.  table, td, th {
7.      border: 2px solid #aaa;
8.  }
9.
10. table {
11.     margin: auto;
12.     border-collapse: collapse;
13.     background-color: rgb(212, 239, 248);
14.     width: 98%;
15. }
16.
17. thead tr th {
18.     background: #B8C1C8;
19.     position: sticky;
20.     top: -2px;
21. }
22.
23. tbody {
24.     overflow-y: scroll;
25. }
26.
27. th {
28.     height: 30px;
29. }
30.
31. td {
32.     text-align: center;
33.     max-width: 100px;
34. }
35.
36. .seq {
37.     word-wrap: break-word;
38.     word-break: break-all;
39.     text-align: left;
40. }
41.
42. div {
43.     margin: auto;
44.     text-align: center;
45. }
46.
47. .container {
48.     width: 80%;
49.     max-height: 800px;
50.     overflow-y: scroll;
51. }
52.
53. tr:hover {
54.     background-color: rgb(228, 250, 234);

```

55. }

10.8 Alfalfa_Database.java

```

1. import java.io.File;
2. import java.sql.*;
3. import org.sqlite.JDBC;
4. import java.util.Arrays;
5. import java.util.HashSet;
6. import java.util.Scanner;
7.
8. public class database {
9.     // given a DNA seq, return its conjugate seq
10.    private static String get_conjugate(String seq) {
11.        String ans = "";
12.        for (int i = seq.length() - 1; i >= 0; i--) {
13.            if (seq.charAt(i) == 'A')
14.                ans += 'T';
15.            else if (seq.charAt(i) == 'T')
16.                ans += 'A';
17.            else if (seq.charAt(i) == 'C')
18.                ans += 'G';
19.            else if (seq.charAt(i) == 'G')
20.                ans += 'C';
21.        }
22.        return ans;
23.    }
24.
25.    public static void main(String[] args) {
26.        try {
27.            // database setup
28.            Scanner sc = new Scanner(System.in);
29.            String dbPath = "./Alfalfa.db";
30.            File dbFile = new File(dbPath);
31.            while (!dbFile.exists()) {
32.                System.out.println("*** Alfalfa.db is not found under the current path: ***");
33.                System.out.println("*** " + dbPath + " ***");
34.                System.out.println("*** Please enter a correct path to Alfalfa.db: ***");
35.                System.out.print("> ");
36.                dbPath = sc.nextLine();
37.                while (!dbPath.contains("/Alfalfa.db")) {
38.                    System.out
39.                        .println("*** The path, absolute or relative, should ends by \"/Alfalfa.db\" ***");
40.                    System.out.print("> ");
41.                    dbPath = sc.nextLine();
42.                }
43.                dbFile = new File(dbPath);
44.            }
45.            String sql = "jdbc:sqlite:" + dbPath;
46.            Class.forName("org.sqlite.JDBC");
47.            Connection conn = DriverManager.getConnection(sql);
48.            Statement stat = conn.createStatement();
49.            ResultSet rs;
50.            int cnt = 0;
51.
52.            // query app
53.            while (true) {
54.                // repeat these prompts after each query until option C is selected
55.                System.out
56.                    .println("*****");

```

```

57. System.out.println("Welcome to the Alfalfa database, please select an option:");
58. System.out.println("[A] Query about all information in Alfalfa.db related to a gene ID");
59. System.out.println("[B] Query about IDs of all genes whose annotations include a keywords");
60. System.out.println("[C] Query about all variations between two points on a chromosome");
61. System.out.println("[D] Exit");
62. System.out.print("> ");
63. String option = sc.nextLine();
64.
65. // exit app
66. if (option.substring(0, 1).toUpperCase().equals("D")) {
67.     System.out.println("Thank you for using the Alfalfa database.");
68.     System.out
69.     .println("*****");
70.     break;
71. }
72.
73. // query mode A
74. if (option.substring(0, 1).toUpperCase().equals("A")) {
75.     System.out.println("Please enter a ID:");
76.     System.out.print("> ");
77.     String query_id = sc.nextLine();
78.
79.     // check validity of query_id, process it if needed
80.     String extension; // "" if no extension
81.     extension = "";
82.     if (query_id.length() == 16) {
83.         query_id = query_id.substring(0, 13); // all records in gff have .01
84.         // extension = "";
85.     } else if (query_id.length() == 20) {
86.         // separate main ID and its extension
87.         extension = query_id.substring(13);
88.         query_id = query_id.substring(0, 13);
89.     } else if (query_id.length() != 13) {
90.         System.out.println("*** The ID should have 13, 16, or 20 characters ***");
91.         continue;
92.     }
93.
94.     // query annotate_info
95.     rs = stat.executeQuery(
96.         "SELECT * FROM annotate_info WHERE ID_full LIKE '\"' + query_id + extension + \"%\\\"";
97.     cnt = 0;
98.     while (rs.next()) {
99.         if (cnt == 0)
100.            System.out.println("*** Results from annotate_info: ***");
101.         cnt++;
102.         System.out.print("id_full = " + rs.getString("id_full") + ", ");
103.         System.out.print("db_name = " + rs.getString("db_name") + ", ");
104.         System.out.print("info1 = " + rs.getString("info1") + ", ");
105.         System.out.println("info2 = " + rs.getString("info2"));
106.     }
107.     if (cnt > 0)
108.         System.out.println("*** " + cnt + " related annotate_info were returned above ***\n");
109.
110.     // query go_info, interpre_info, kegg_info, pathway_info, and pfam_info
111.     rs = stat.executeQuery(
112.         "SELECT pros_info.ID_full, pros_info.unique_id, kegg_info.kegg, pathway_info.pathway,"
113.         + " go_info.go, interpre_info.interpro, pfam_info.pfam FROM pros_info\n"
114.         + " LEFT OUTER JOIN go_info ON pros_info.ID_full = go_info.ID_full\n"
115.         + " LEFT OUTER JOIN interpre_info ON pros_info.ID_full = interpre_info.ID_full\n"
116.         + " LEFT OUTER JOIN kegg_info ON pros_info.ID_full = kegg_info.ID_full\n"
117.         + " LEFT OUTER JOIN pathway_info ON pros_info.ID_full = pathway_info.ID_full\n"

```

```

118.         + " LEFT OUTER JOIN pfam_info ON pros_info.ID_full = pfam_info.ID_full\n"
119.         + " WHERE pros_info.ID_full LIKE \"\" + query_id + extension + \"%\";\";
120.     cnt = 0;
121.     while (rs.next()) {
122.         if (rs.getString("go") != null || rs.getString("interpro") != null
123.             || rs.getString("kegg") != null || rs.getString("pathway") != null
124.             || rs.getString("pfam") != null) {
125.             if (cnt == 0)
126.                 System.out.println("*** Annotations from other databases: ***");
127.             cnt++;
128.             System.out.println("< id full = " + rs.getString("id full") + " >");
129.             System.out.println("go = " + rs.getString("go"));
130.             System.out.println("interpro = " + rs.getString("interpro"));
131.             System.out.println("kegg = " + rs.getString("kegg"));
132.             System.out.println("pathway = " + rs.getString("pathway"));
133.             System.out.println("pfam = " + rs.getString("pfam"));
134.         }
135.     }
136.     if (cnt > 0)
137.         System.out
138.             .println("*** Related annotations from other databases were returned above ***\n");
139.
140.     // query pros info
141.     rs = stat.executeQuery(
142.         "SELECT * FROM pros_info WHERE ID_full LIKE \"\" + query_id + extension + \"%\";\";
143.     cnt = 0;
144.     while (rs.next()) {
145.         if (cnt == 0)
146.             System.out.println("*** Results from pros_info database: ***");
147.         cnt++;
148.         System.out.print("id full = " + rs.getString("id full") + ", ");
149.         System.out.println("pros = " + rs.getString("pros"));
150.     }
151.     if (cnt > 0)
152.         System.out.println("*** " + cnt + " related pros info were returned above ***\n");
153.
154.     // query gff
155.     rs = stat.executeQuery("SELECT * FROM gff AS G, ID_to_uniqueid AS I WHERE I.ID = \"\"
156.         + query_id + "\" AND G.unique_id >= I.start AND G.unique_id <= I.end;\";
157.     cnt = 0;
158.     while (rs.next()) {
159.         if (!extension.equals("")) {
160.             String full_id = query_id + extension;
161.             String rst_id = rs.getString("attributes");
162.             if (!rst_id.contains(full_id))
163.                 continue;
164.         }
165.         if (cnt == 0)
166.             System.out.println("*** Results from gff: ***");
167.         cnt++;
168.         System.out.print("seqid = " + rs.getString("seqid") + ", ");
169.         System.out.print("source = " + rs.getString("source") + ", ");
170.         System.out.print("type = " + rs.getString("type") + ", ");
171.         System.out.print("start = " + rs.getString("start") + ", ");
172.         System.out.print("end = " + rs.getString("end") + ", ");
173.         System.out.print("score = " + rs.getString("score") + ", ");
174.         System.out.print("strand = " + rs.getString("strand") + ", ");
175.         System.out.print("phase = " + rs.getString("phase") + ", ");
176.         System.out.print("attributes = " + rs.getString("attributes") + ", ");
177.         String dna = rs.getString("DNA");
178.         // replace by DNA conjugates when needed

```



```

179.         if (rs.getString("strand").equals("-"))
180.             dna = get_conjugate(dna);
181.         System.out.println("DNA = " + dna);
182.     }
183.     if (cnt > 0)
184.         System.out.println("*** " + cnt
185.             + " related gff records and their DNA sequences were returned above ***");
186.     else
187.         System.out.println("*** NO related information was found in the Alfalfa database ***");
188.     rs.close();
189. }
190.
191. // query mode B
192. if (option.substring(0, 1).toUpperCase().equals("B")) {
193.     System.out.println("Please enter some keywords:");
194.     System.out.print("> ");
195.     String keywords = sc.nextLine();
196.     rs = stat.executeQuery("SELECT * FROM annotate_info WHERE info1 LIKE \"%\" + keywords
197.         + \"%\" OR info2 LIKE \"%\" + keywords + \"%\"");
198.     cnt = 0;
199.     int number_of_unique_id = 0;
200.     HashSet<String> my_set = null;
201.     while (rs.next()) {
202.         if (cnt == 0) {
203.             my_set = new HashSet<String>();
204.             number_of_unique_id = 0;
205.         }
206.         cnt++;
207.         if (my_set.add(rs.getString("id_full")))
208.             number_of_unique_id++;
209.         System.out.print("id_full = " + rs.getString("id_full") + ", ");
210.         System.out.print("db_name = " + rs.getString("db_name") + ", ");
211.         System.out.print("info1 = " + rs.getString("info1") + ", ");
212.         System.out.println("info2 = " + rs.getString("info2"));
213.     }
214.     if (cnt > 0) {
215.         System.out.println("*** " + cnt + " related annotate_info were returned above ***");
216.         System.out.println("*** " + number_of_unique_id + " unique gene IDs involved: ***");
217.         if (my_set != null && !my_set.isEmpty())
218.             System.out.println(my_set.toString());
219.     } else
220.         System.out.println("*** NO related information was found in the Alfalfa database ***");
221.     rs.close();
222. }
223.
224. // query mode C
225. if (option.substring(0, 1).toUpperCase().equals("C")) {
226.     System.out.println("Please enter a chromosome number (an integer between 1 and 8):");
227.     System.out.print("> ");
228.     int num;
229.     try {
230.         num = Integer.parseInt(sc.nextLine());
231.         if (num < 1 || num > 8) {
232.             System.out.println("*** The chromosome number entered was invalid ***");
233.             continue;
234.         }
235.     } catch (NumberFormatException e) {
236.         System.out.println("*** The chromosome number entered was invalid ***");
237.         continue;
238.     }
239. }

```

```

240. // values from get_max_vcfID.sql and get_min_vcfIDs.sql
241. int max_pos = 148124263;
242. int arr[] = {1, 17551074, 32991065, 52346699, 69388115, 89626087, 113452234, 132015073,
243.     max_pos + 1};
244.
245. int vcf_ID_lower_bound = arr[num - 1];
246. int vcf_ID_upper_bound = arr[num] - 1;
247. int POS_lower_bound;
248. int POS_upper_bound;
249.
250. try {
251.     System.out.println("Please enter a lower bound of POS (a positive integer):");
252.     System.out.print("> ");
253.     POS_lower_bound = Integer.parseInt(sc.nextLine());
254.     if (POS_lower_bound < 1) {
255.         System.out.println("*** The lower bound entered was invalid ***");
256.         continue;
257.     }
258.
259.     System.out.println("Please enter an upper bound of POS (a positive integer):");
260.     System.out.print("> ");
261.     POS_upper_bound = Integer.parseInt(sc.nextLine());
262.     if (POS_upper_bound < 1) {
263.         System.out.println("*** The upper bound entered was invalid ***");
264.         continue;
265.     }
266. } catch (NumberFormatException e) {
267.     System.out.println("*** The entered String was not an Integer ***");
268.     continue;
269. }
270.
271. // query vcf
272. rs = stat.executeQuery("SELECT * FROM vcf WHERE vcf_ID BETWEEN " + vcf_ID_lower_bound
273.     + " AND " + vcf_ID_upper_bound + " AND POS BETWEEN " + POS_lower_bound + " AND "
274.     + POS_upper_bound + ";");
275.
276. cnt = 0;
277. ResultSetMetaData rsmd = rs.getMetaData();
278. int columnsNumber = rsmd.getColumnCount();
279. while (rs.next()) {
280.     cnt++;
281.     for (int i = 1; i <= columnsNumber; i++) {
282.         if (i > 1)
283.             System.out.print(", ");
284.         String columnValue = rs.getString(i);
285.         System.out.print(rsmd.getColumnName(i) + " = " + columnValue);
286.     }
287.     System.out.println();
288. }
289. System.out.println("*** " + cnt + " related VCF records were returned above ***");
290. }
291. }
292.
293. // close database
294. sc.close();
295. conn.close();
296. } catch (Exception e) {
297.     e.printStackTrace();
298. }
299. }
300. }

```


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 & 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., & 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>.