Exploring and Comparing Database Modeling on Open Target Genetics Data from SQL to noSQL

Zhang, Wenliang

CS779 Advanced Database Management

2021 Fall Term Project

Abstract

Open Targets is a public-private partnership that aims to support for systematic drug target discovery and prioritization. It has created a public platform that integrates large collections of big datasets from human genetics and genomics studies, which has been served as a very useful data sources for drug development. However, the size and complexity of the whole genomic data has posed a big challenge for scientists to do data query and computation. In this term project, we have explored and compared a few database techniques from SQL to noSQL for modeling on the Open Target genetics data in order to provide prototyping and proof-of-concept of various data modeling and queries. We have developed a simple web application and used subsets of the data to compare the data ETL process and the response time of the data query. The goal is to demonstrate the feasibility and choices of database modeling in order to design and scale up the more demanding queries necessary for the complete big data sets.

TABLE OF CONTENTS

1. INTRODUCTION TO OPEN TARGET DATASETS

1.1 OVERVIEW

Open Targets is a public-private partnership between industries and academia that aims to help life-science research scientists to identify and prioritize potential therapeutic drug targets. It has created a comprehensive open-source research platform that integrates a large range of publicly available datasets generated from human genetics and genomics studies.

There are two main data resources provided by Open Targets: **Open Target Platform** and **Open Targets Genetics Portal**. Open Targets Platform data focuses more on the broad range of biological data that are curated and summarized as evidence of target-disease association, with a score ranging from 0-1, while Open Targets Genetics Portal focuses on Genome-Wide-Association-Study (GWAS) and functional genomics data to enable large scale exploration and identification of causal variants and genes. It includes the disease-agnostic Variant-to-Gene (V2G) mapping and a disease-specific Locus-to-Gene (L2G) mapping for trait-associated loci, both of which summarize the results as a score ranging from 0-1. It also includes colocalisation analysis between studies and diseases.

1.2 DATA ACCESS AND DESCRIPTION

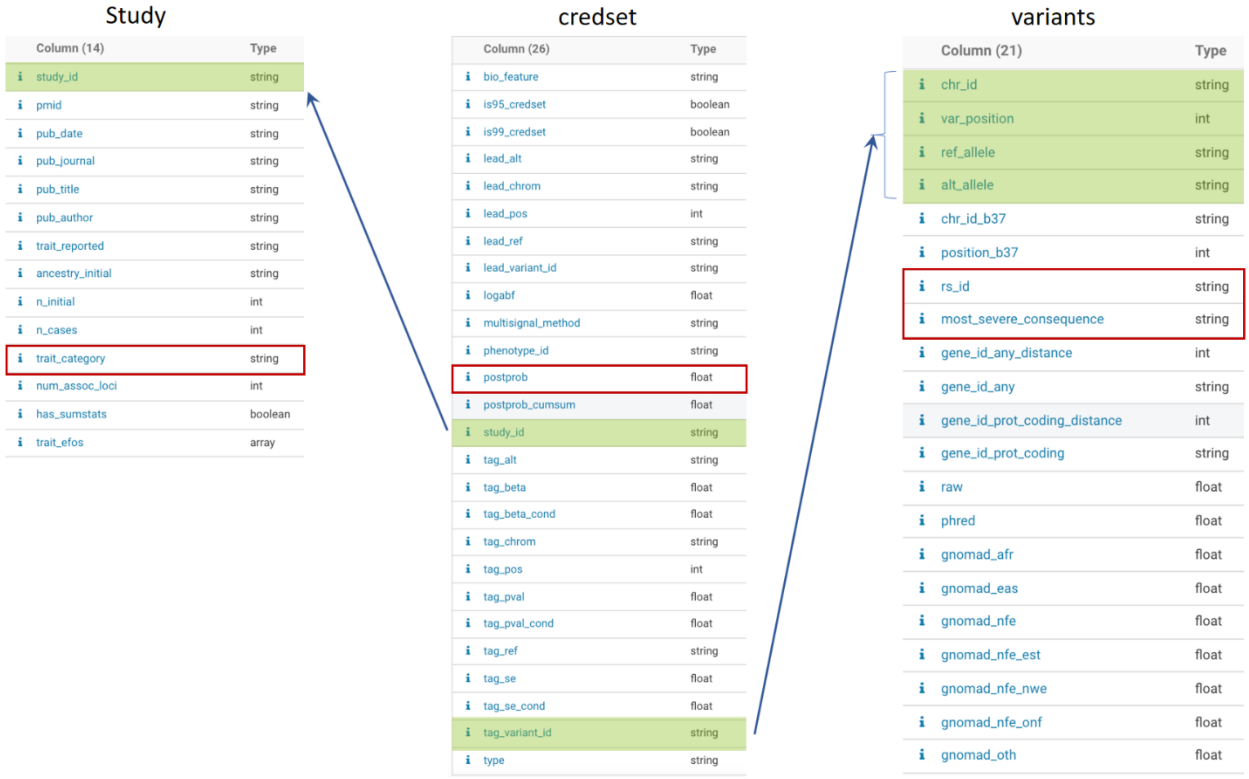
To access the Open Targets data, you can query data either through the web interface manually or access through their API programmatically. However, both methods are not ideal for doing analysis at a very large scale, where the queries are getting bigger and more complex and may require the whole data table from the database. So in order to harness the data better being able to query data in a more flexible way and at a larger scale, I decided to download the whole datasets, and explore various options for database modeling. The current version of the whole Open Target Genetics data size is about 1.2 Terabytes, mostly in JSON and parquet format. The description, size and format information of the complete Open Target Genetics datasets are listed in the table as below:

Dataset	Description	Size(Format)
d2v2g	Intersection table linking studies to variants to genes	700 Gb (JSON)
l2g	Locus to gene scores and subscores	500 Mb (csv)
sa_gwas	Summary statistics for GWAS studies	120 Gb (parquet)
sa_molecular_trait	Summary statistics for molecular trait studies	35 Gb (parquet)
v2d	Data linking variants to studies	10 Gb (JSON)
v2d_coloc	GWAS-GWAS and GWAS-molecular trait colocalisation results	2 Gb (JSON)
v2d_credset	Fine mapped credible sets for all GWAS	2 Gb (JSON)
v2g	Data linking variants to genes from V2G pipeline	251 Gb (JSON)
variant-index	Variant index	45 Gb (JSON)
genes-index	Gene index	12 Mb (JSON)
study-index	Study metadata including trait, publication and ancestry information	10 Mb (JSON)
overlap-index	Pair-wise overlap between all independently associated GWAS loci	1 Gb (JSON)

1.3 SAMPLE DATASETS USED FOR PROTOTYPING

In order to prototype and compare the database modeling, in this project I have chosen three datasets, v2d_credset, variant-index, and study-index for my database modeling process. The study table contains various information about each GWAS Catalog study. Each study has a unique study_id. The credset is the finemapping table containing Credible set anlaysis results used to link index (or lead) variants to tag variants. Variants table has all the information about each variant.

The information about each dataset is listed as below:



The complete datasets are used for the implementation on the big data Cloudera platform while only the subsets of the three datasets were used as the test datasets for prototyping on other database software platforms in order to speed up the iteration of the development cycle. The subletting process is done on the HDFS on Cloudera platform using Hive (For details, please refer to session 2.1).

A data query is designed to validate the performance of the joined operations on each platform based on the time cost to get the query results. The objective of the query is to find coding variants with high posterior probability for a certain trait category of interest.

2. EXPLORE AND COMPARE DATABASE MODELING

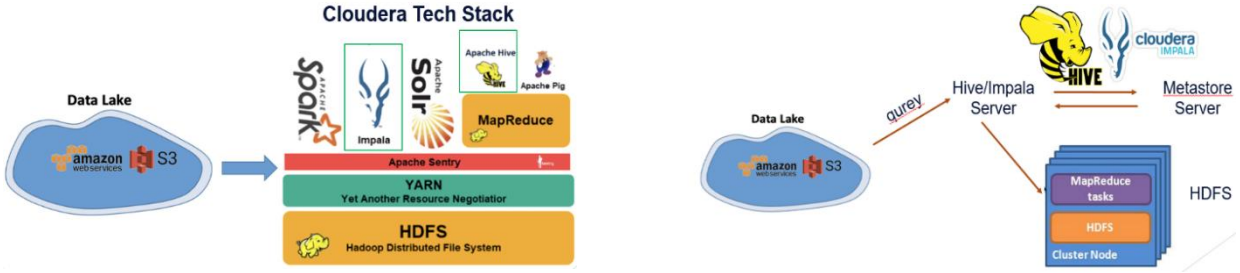
The following database software platforms have been explored in this project. We have compared the data modeling, data ETL process as well as the performance of the designed joined query. The platforms include the following:

1) Big data platform, Cloudera Hadoop HDFS system with Hive and Impala as the query engines
2) DBMS PostgreSQL
3) A newly emerged array-based database called TileDB
4) Document based noSQL solution using MongoDB
5) Lastly the combined solution using MongoDB and pySpark

2.1 CLOUDERA WITH HIVE AND IMPALA

Cloudera is the commercial version of Hadoop ecosystem with many useful big data tools such as, Spark, Hive and Impala. The biggest advantage of using Cloudera with Hive and Impala is that you can use the schema on the read, which means that you don't have to design the schema beforehand. That is to say that we can save all the raw data in the data lake, for example, AWS S3, and then design the schema and write meta into HDFS meta store.

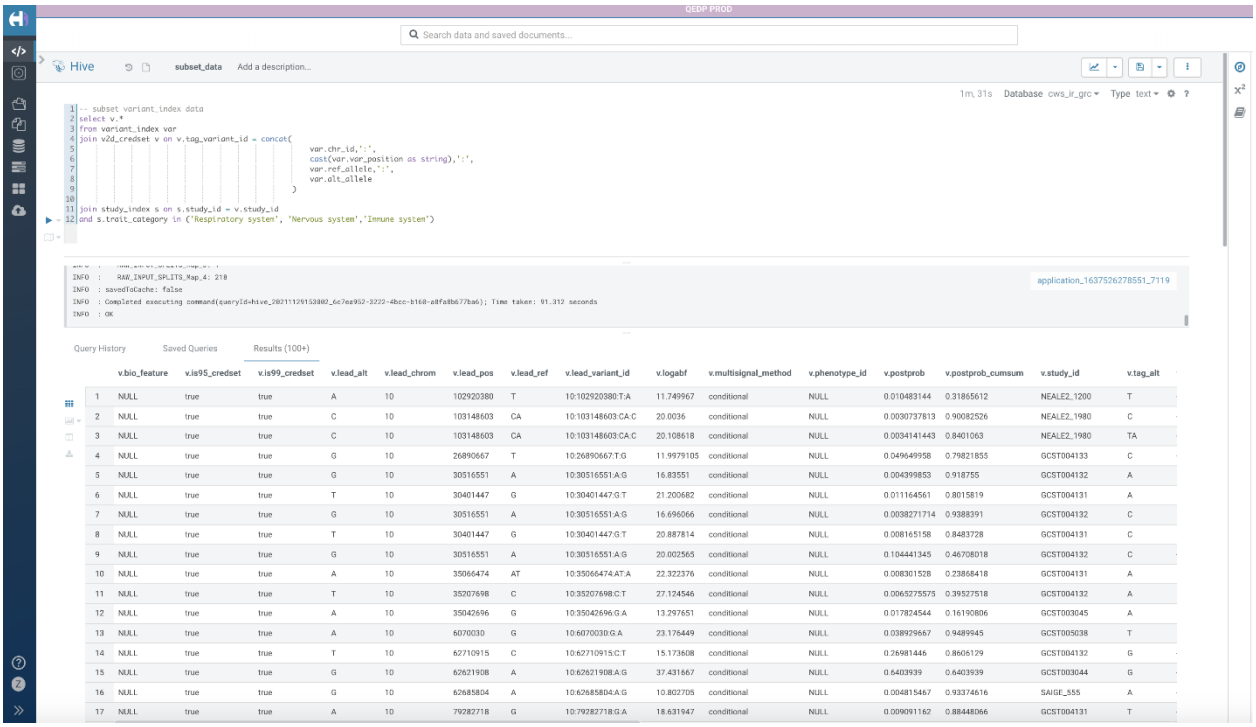
Hive and Impala are the tools that allow you to write SQL like script to do interactive big data query from a distributed data system. The language of Hive and Impala is very like SQL, so you don't have to learn a completely new language in order to use. Behind the hood, Hive is using the map-reduce jobs that are executed and processed on the Hadoop system while Impala has its own algorithm of processing the data. Compared to Hive, Impala generally performs faster but more memory intensive and does not support fault tolerant as it is not using the map-reduce algorithm like Hive does. Also Hive support more complex data type. Therefore, Hive is more suitable for batch data query and processing, while Impala is more appropriate for interactive data query and exploration or small datasets processing. Here below is the diagram of the architecture summarizing how the Hive and Impala query works.



Here is the example query to get variant and study information for the credset of high posterior probability.

```sql
-- select high posterior porb, join study_id with v2d_credset tables
select
    s.study_id,
    s.trait_category,
    s.trait_efos,
    v.postprob
from study_index s
join v2d_credset v on s.study_id = v.study_id
where v.postprob > 0.5
and s.trait_category = 'Respiratory system'
```

In order to explore and prototype using other database management system, the following HIV SQL script was used to subset the three dataset by filtering the data on the column of trait_category of the study table to contain only the three categories: Respiratory System, Immune System and Nervous System. The query is performed on the HUE (Hadoop User Experience). Below is the screenshot of the script and the query results. The query takes about one and half minutes to finish. After subset and data cleaning, we are able to reduce the size of test dataset significantly.



Study_index: 19267 records
Variant_index: 72878709 records
V2d_credset: 31694678 records

Study_index_sample: 202 records
Variant_index_sample: 43626 records
V2d_credset_sample: 68362 records

2.2 POSTGRESQL

We started to model the test datasets using the relational database management system, PostgreSQL and used it as benchmark for evaluating query performance. The ETL process was done using Python Pandas and sqlalchemy packages. We read the sample dataset in csv format and inject data into the PostgreSQL database after some data cleaning. The codes are shown below:

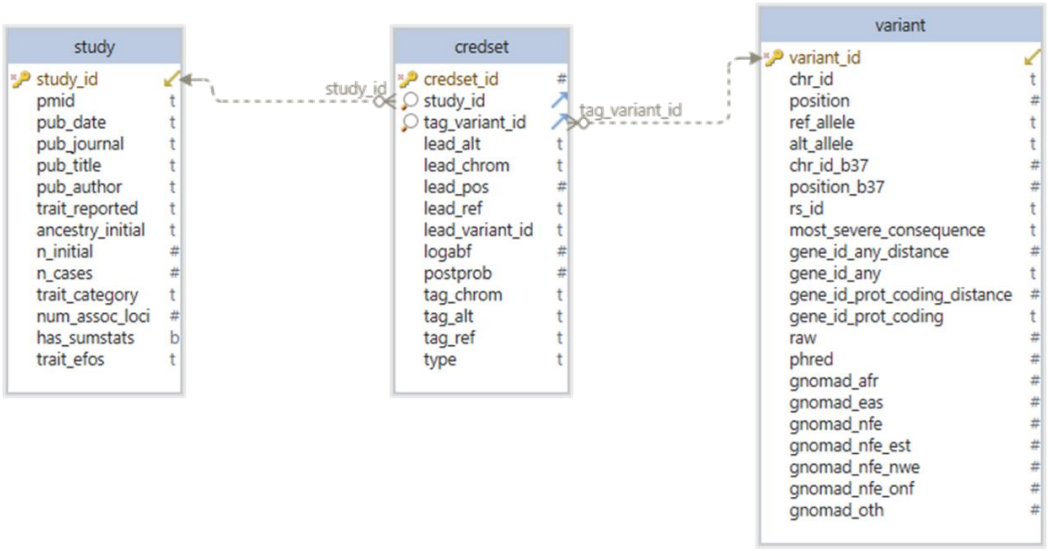
```
conn_string = 'postgresql://opentarget:opentarget@localhost/opentarget'
engine = create_engine(conn_string)

# credset sample
credset_df = pd.read_csv('data/credset_sample.csv')
credset_cols_keep = ['lead_alt', 'lead_chrom', 'lead_pos', 'lead_ref',
'lead_variant_id', 'logabf', 'postprob',
                     'study_id', 'tag_chrom','tag_alt', 'tag_ref',
'tag_variant_id', 'type']
credset_df = credset_df[credset_cols_keep]
credset_df = credset_df.reset_index()
credset_df['tag_chrom'] = credset_df['tag_chrom'].astype('str')
credset_df['lead_chrom'] = credset_df['lead_chrom'].astype('str')
credset_df.to_sql('credset', con=engine, if_exists='replace',index=False)

# study_sample
study_df = pd.read_csv('data/study_sample.csv')
study_df = study_df.reset_index()
study_df.to_sql('study', con=engine, if_exists='replace',index=False)

# Variant_sample
variant_df = pd.read_csv('data/variant_sample.csv')
variant_df = variant_df.reset_index()

variant_df['chr_id'] = variant_df['chr_id'].astype('str')
variant_df['variant_id'] = variant_df.apply(lambda x:
':'.join([x['chr_id'],str(x['position']),x['ref_allele'],x['alt_allele']]),axis=1)
```

Then the primary and Foreign keys and Index were created. The viaraint_id field was created and used as the primary key for the variant The code is as below:

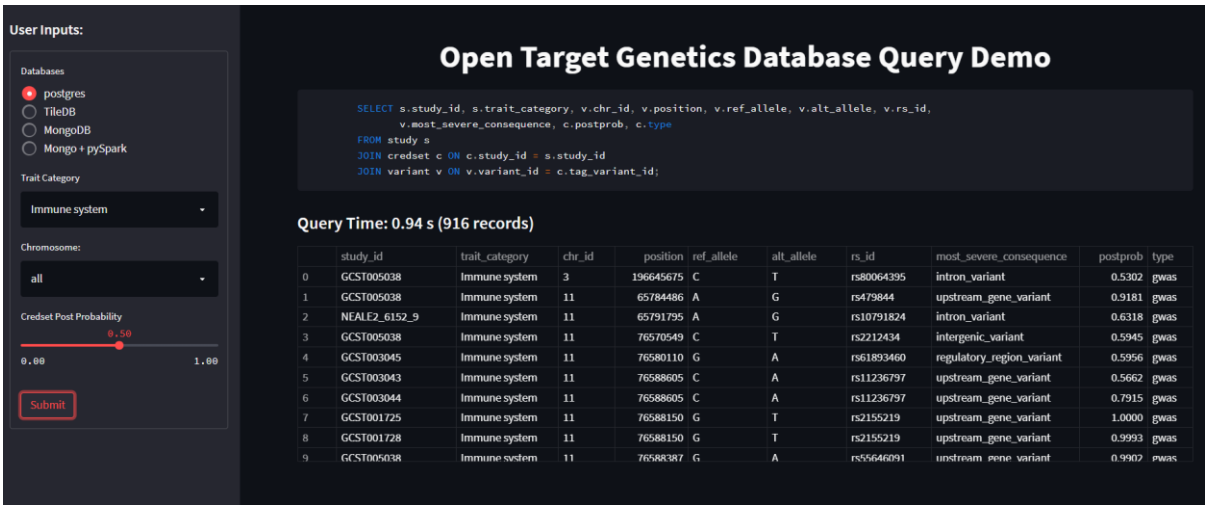
```
variant_df['variant_id'] = variant_df.apply(lambda x:
':'.join([x['chr_id'],str(x['position']),x['ref_allele'],x['alt_allele']]),axis=1)
```

```
ALTER TABLE study
ADD PRIMARY KEY (study_id);

ALTER TABLE variant
--ADD PRIMARY KEY (chr_id, position ,ref_allele, alt_allele);
ADD PRIMARY KEY (variant_id);

ALTER TABLE credset
ADD COLUMN credset_id SERIAL PRIMARY KEY;

ALTER TABLE credset
ADD FOREIGN KEY (study_id) REFERENCES study (study_id),
ADD FOREIGN KEY (tag_variant_id) REFERENCES variant (variant_id);

CREATE INDEX idx_credset_study_id on credset(study_id);
CREATE INDEX idx_credset_tag_variant_id on credset(tag_variant_id);
```

After we created the primary and foreign keys for all three tables, the schema looks like below:



To test the performance of our joined query, we executed the query command using our web application. The query time shows that it is performing very well with less than 1 sec with our test datasets. The snapshot of the web application and query results are shown below:



2.3 TILEDB

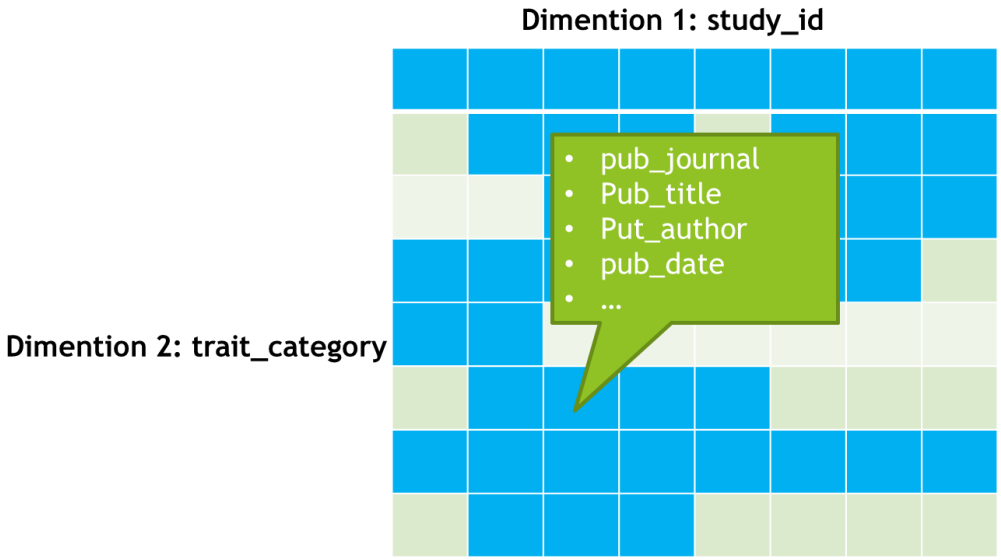
TileDB is a universal data storage engine architected around multi-dimensional arrays that enables storing and accessing data using Dense or Sparse arrays data models. It has the following key features:

- Native cloud object storage and access
- Read/write queries are fully parallelized
- Multi-dimensional indexing
- Query times grow with result not data size

Build study TileDB array using Python API:

```
if tiledb.array_exists(os.path.join('opentarget_arrays', 'study_tldb')):
    tiledb.remove(os.path.join('opentarget_arrays', 'study_tldb'))
    tiledb.from_csv(os.path.join('opentarget_arrays', 'study_tldb'),
                    'data/study_sample.csv',
                    index_dims=['study_id', 'trait_category'],
                    chunksize=10 ** 6,
                    sparse=True,
                    fillna={"trait_efos": '',
                            'ancestry_replication': '',
                            'ancestry_initial': '',
                            'pmid':'',
                            'pub_journal':'',
                            'pub_title':''})
```

Here is the simplified diagram showing how the example study TileDB array looks like. The two attributes that are used for dimensions of the array are indexed for speedy data slicing. The TileDB array looks very much like a cluster indexed view in relational database or multi-dimensional parquet file.



After we build the array, we can read the schema. The code below shows the array schema of the study array we build. We can tell from the schema, the information about the dimensions, domains, attributes, data types, cell and tile order etc.

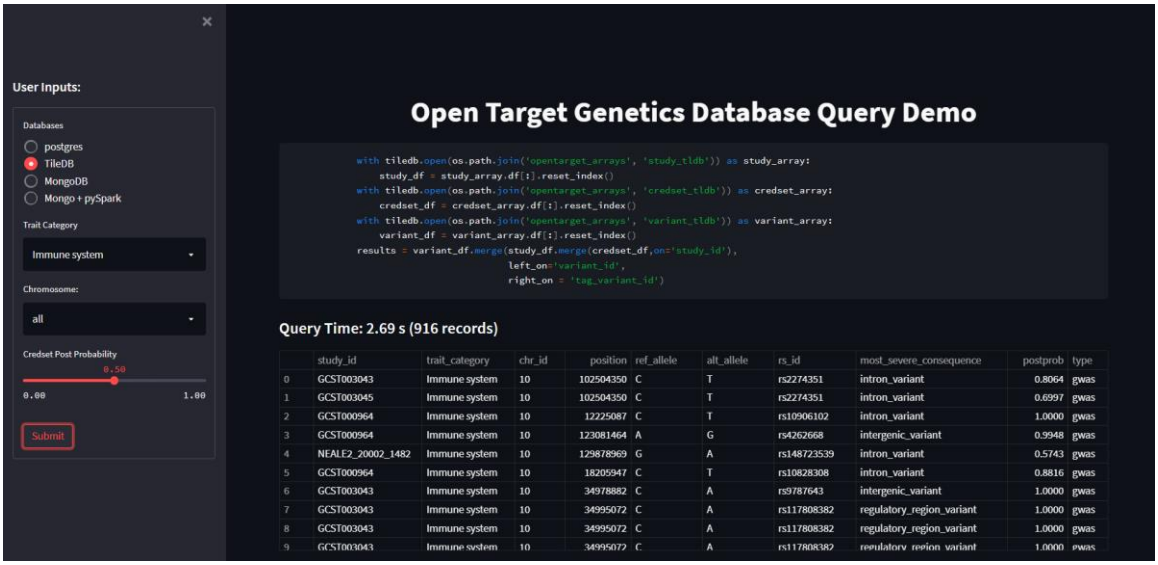
```
In [80]: with tiledb.open('study_tldb') as A:
             study_df = A.df[:]
             study_schema = A.schema

         print(study_schema)
         study_df.head()
```

```
ArraySchema(
  domain=Domain(*[
    Dim(name='study_id', domain=(None, None), tile='None', dtype='|S0', var=True),
    Dim(name='trait_category', domain=(None, None), tile='None', dtype='|S0', var=True),
  ]),
  attrs=[
    Attr(name='pmid', dtype='<U0', var=True, nullable=False),
    Attr(name='pub_date', dtype='<U0', var=True, nullable=False),
    Attr(name='pub_journal', dtype='<U0', var=True, nullable=False),
    Attr(name='pub_title', dtype='<U0', var=True, nullable=False),
    Attr(name='pub_author', dtype='<U0', var=True, nullable=False),
    Attr(name='trait_reported', dtype='<U0', var=True, nullable=False),
    Attr(name='ancestry_initial', dtype='<U0', var=True, nullable=False),
    Attr(name='n_initial', dtype='int64', var=False, nullable=False),
    Attr(name='n_cases', dtype='float64', var=False, nullable=False),
    Attr(name='num_assoc_loci', dtype='int64', var=False, nullable=False),
    Attr(name='has_sumstats', dtype='uint8', var=False, nullable=False),
    Attr(name='trait_efos', dtype='<U0', var=True, nullable=False),
    Attr(name='ancestry_replication', dtype='<U0', var=True, nullable=False),
    Attr(name='n_replication', dtype='float64', var=False, nullable=False),
  ],
  cell_order='row-major',
  tile_order='row-major',
  capacity=10000,
  sparse=True,
  allows_duplicates=True,
  coords_filters=FilterList([ZstdFilter(level=-1)]),
)
```

We can also easily read the data from the array in the format of Pandas dataframe. The syntax used for slicing and reading the array data is very similar to pandas as well. The dimensions of the array will become the index of the dataframe. The code below shows the dataframe read from the study array.

study_id	trait_category	pmid	pub_date	pub_journal	pub_title	pub_author	trait_reported	ancestry_initial	n_initial	n_cases	num_assoc_lc
GCST000001	Nervous system	PMID:15761122	2005-03-10	Science	Complement factor H polymorphism in age-relate...	Klein RJ	Age-related macular degeneration	['European=146']	146	96.0	
GCST000002	Nervous system	PMID:16252231	2005-09-09	Am J Hum Genet	High-resolution whole-genome association study...	Maraganore DM	Parkinson's disease	['Asian unspecified=1', 'European=744', 'Other...	886	NaN	
GCST000003	Disease	PMID:16614226	2006-04-14	Science	A common genetic variant is associated with ad...	Herbert A	Obesity	['NR=694']	694	NaN	
GCST000004	Cardiovascular measurement	PMID:16648850	2006-04-30	Nat Genet	A common genetic variant in the NOS1 regulator...	Arking DE	QT interval	['European=200']	200	NaN	
GCST000005	Nervous system	PMID:17052657	2006-09-28	Lancet Neurol	Genome-wide genotyping in Parkinson's disease ...	Fung HC	Parkinson's disease	['European=537']	537	267.0	

In order to test and compare the performance of our joined query, we tested the query command using our web application. There are multiple ways for doing queries with tiledb based on the documentation and consultation with their technical support. MariaDB tiledb plugin, called MyTile, is one of the choices while it is not tested in here. Pandas and pySpark are the other two choices to perform join operations after the data is read out from the array. Because of the many technical issues I met, I finally choose Python, which is shown in the below example. As we can tell the query time is pretty fast with our test datasets. The snapshot of the web application and query results are shown below:



	study_id	trait_category	chr_id	position	ref_allele	alt_allele	rs_id	most_severe_consequence	postprob	type
0	GCST003043	Immune system	10	102504350	C	T	rs2274351	intron_variant	0.8064	gwas
1	GCST003045	Immune system	10	102504350	C	T	rs2274351	intron_variant	0.6997	gwas
2	GCST000964	Immune system	10	12225087	C	T	rs10906102	intron_variant	1.0000	gwas
3	GCST000964	Immune system	10	123081464	A	G	rs4262668	intergenic_variant	0.9948	gwas
4	NEALE2_20002_1482	Immune system	10	129878969	G	A	rs148723539	intron_variant	0.5743	gwas
5	GCST000964	Immune system	10	18205947	C	T	rs10828308	intron_variant	0.8816	gwas
6	GCST003043	Immune system	10	34978882	C	A	rs9787643	intergenic_variant	1.0000	gwas
7	GCST003043	Immune system	10	34995072	C	A	rs117808382	regulatory_region_variant	1.0000	gwas
8	GCST003043	Immune system	10	34995072	C	A	rs117808382	regulatory_region_variant	1.0000	gwas
9	GCST003043	Immune system	10	34995072	C	A	rs117808382	regulatory_region_variant	1.0000	gwas

2.4 MONGODB

Next, I considered to try the document based noSQL database, MongoDB. The data inject process is proved to an easy process as inspected. Python MongoClient of Pymongo package is used to inject the data in the csv format into MongoDB. Below shows the snapshot of the data injection process.

```python
def mongoimport(client, csv_path, db_name, coll_name):
    db = client[db_name]
    coll = db[coll_name]
    data = pd.read_csv(csv_path)
    payload = json.loads(data.to_json(orient='records'))
    coll.delete_many({})
    coll.insert_many(payload)
    return coll.count_documents({})

# # Create database and three collections
client = MongoClient("mongodb://localhost:27017/")
db = client["opentarget"]
study=db.study
variant=db.variant
credset=db.credset

mongoimport(client, 'data/study_sample.csv', 'opentarget','study')
mongoimport(client, 'data/credset_sample.csv', 'opentarget','credset')
mongoimport(client, 'data/variant_sample_withkey.csv', 'opentarget','variant')

# create index to improve join performance
credset.create_index('study_id')
credset.create_index('tag_variant_id')
```

Here is the what the three documents looks like after data injection.

variant

```
{'_id': ObjectId('619c6f040bf566f731cd1f75'),
 'alt_allele': 'G',
 'chr_id': 4,
 'chr_id_b37': 4,
 'gene_id_any': 'ENSG00000248115'
 'gene_id_any_distance': 165445,
 'gene_id_prot_coding': 'ENSG0000
 'gene_id_prot_coding_distance':
 'gnomad_afr': 0.001148106,
 'gnomad_eas': 0.0,
 'gnomad_nfe': 0.00473227,
 'gnomad_nfe_est': 0.002394428,
 'gnomad_nfe_nwe': 0.006169965,
 'gnomad_nfe_onf': 0.004213483,
 'gnomad_oth': 0.004604051,
 'index': 0,
 'most_severe_consequence': 'intr
 'phred': 3.122,
 'position': 53111094,
 'position_b37': 53977261,
 'raw': 0.031534,
 'ref_allele': 'A',
 'rs_id': 'rs574753165',
 'variant_id': '4:53111094:A:G'}
```

credset

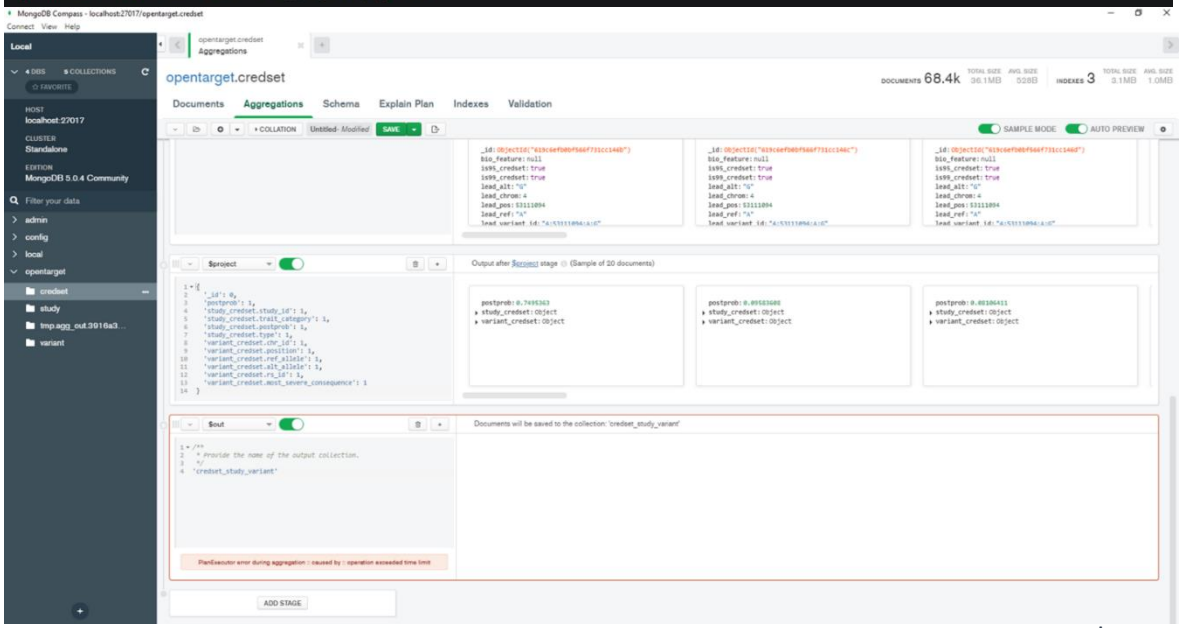
```
{'_id': ObjectId('619c6efb0bf566f731cc146b'),
 'bio_feature': None,
 'is95_credset': True,
 'is99_credset': True,
 'lead_alt': 'G',
 'lead_chrom': 4,
 'lead_pos': 53111094,
 'lead_ref': 'A',
 'lead_variant_id': '4:53111094:A:G',
 'logabf': 14.310652,
 'multisignal_method': 'conditional',
 'phenotype_id': None,
 'postprob': 0.7495363,
 'postprob_cumsum': 0.74
 'study_id': 'GCST004695
 'tag_alt': 'G',
 'tag_beta': 0.39474,
 'tag_beta_cond': 0.3947
 'tag_chrom': 4,
 'tag_pos': 53111094,
 'tag_pval': 5e-09,
 'tag_pval_cond': 5e-09,
 'tag_ref': 'A',
 'tag_se': 0.0675027,
 'tag_se_cond': 0.067502
 'tag_variant_id': '4:53
 'type': 'gwas'}
```

study

```
{'_id': ObjectId('619c6ef50bf566f731cc13a1'),
 'ancestry_initial': '["European=113006"]',
 'has_sumstats': True,
 'n_cases': 32384.0,
 'n_initial': 113006,
 'num_assoc_loci': 2,
 'pmid': 'PMID:28604731',
 'pub_author': 'Hammerschlag AR',
 'pub_date': '6/12/17',
 'pub_journal': 'Nat Genet',
 'pub_title': 'Genome-wide association analysis of insomnia complaints '
              'identifies risk genes and genetic overlap with psychiatric and '
              'metabolic traits.',
 'study_id': 'GCST004695',
 'trait_category': 'Nervous system',
 'trait_efos': '["EFO_0004698"]',
 'trait_reported': 'Insomnia complaints'}
```

Because MongoDB is not a relational database, we cannot perform traditional join operations like in SQL database. However, you can perform a left outer join by using the $lookup stage. The $lookup stage lets you specify which collection you want to join with the current collection, and which fields that should match. In this test join query, I have to do two lookup stage as the code shows below. However, I was not able to write out the joined results, but saved in a generator for iterations.

```
pipeline = [{'$lookup':{'from': 'study','localField': 'study_id','foreignField':'study_id','as': 'study_credset'}},
            {'$unwind': '$study_credset'},
            {'$lookup':{'from': 'variant','localField':'tag_variant_id','foreignField': 'variant_id','as': 'variant_credset'}},
            {'$unwind': '$variant_credset'},
            {'$project':
                    {'_id': 0, 'postprob': 1, 'study_credset.study_id': 1, 'study_credset.trait_category': 1,
                     'study_credset.postprob': 1, 'study_credset.type': 1, 'variant_credset.chr_id': 1,
                     'variant_credset.position': 1, 'variant_credset.ref_allele': 1, 'variant_credset.alt_allele': 1,
                     'variant_credset.rs_id': 1, 'variant_credset.most_severe_consequence': 1}}]

joined_collection = (credset.aggregate(pipeline))
```

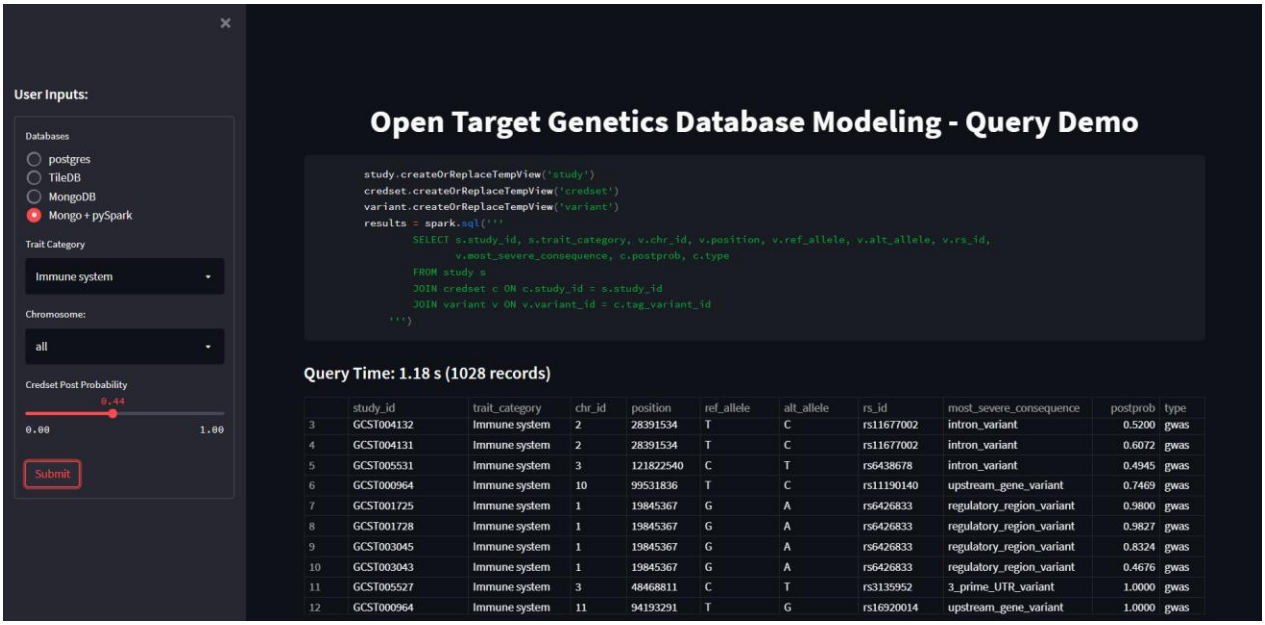


2.5 MONGODB + SPARK

Due to the difficulties and slow speed of doing joined operations within MongoDB, I decided to just use the MongoDB for storage data and use Spark to read data from MongoDB and perform joined operations. Here below is the code shows how to read data from MongoDB into Spark DataFrame:

```
# Read Mongo collection into pySpark Dataframe
study = spark.read.format('com.mongodb.spark.sql.DefaultSource')\
             .option('uri','mongodb://127.0.0.1/opentarget.study')\
             .load()
credset = spark.read.format('com.mongodb.spark.sql.DefaultSource')\
             .option('uri','mongodb://127.0.0.1/opentarget.credset')\
             .load()
variant = spark.read.format('com.mongodb.spark.sql.DefaultSource')\
             .option('uri','mongodb://127.0.0.1/opentarget.variant')\
             .load()
```

To do the joined query in pySpark, we first need to create temp views for each of the Spark Dataframes. Then we can construct SQL string of the query exactly the same way as in relational database. The spark will be able to turn the SQL into its own execution plan under the hood and save the results into a Spark dataframe. Because of the distributed environment, the data is partitioned into many partitions and the execution of the joined SQL plan will evolve a lot of data shuffling which will lead some overhead problem when we write data into the console. Therefore, we see the execution time is a little bit slow. However, because of the distributed computing under the hood, we will be able to process join operation between very large tables. So the advantage of this method is that the join query will be very scalable.



	study_id	trait_category	chr_id	position	ref_allele	alt_allele	rs_id	most_severe_consequence	postprob	type
3	GCST004132	Immune system	2	28391534	T	C	rs11677002	intron_variant	0.5200	gwas
4	GCST004131	Immune system	2	28391534	T	C	rs11677002	intron_variant	0.6072	gwas
5	GCST005531	Immune system	3	121822540	C	T	rs6438678	intron_variant	0.4945	gwas
6	GCST000964	Immune system	10	99531836	T	C	rs11190140	upstream_gene_variant	0.7469	gwas
7	GCST001725	Immune system	1	19845367	G	A	rs6426833	regulatory_region_variant	0.9800	gwas
8	GCST001728	Immune system	1	19845367	G	A	rs6426833	regulatory_region_variant	0.9827	gwas
9	GCST003045	Immune system	1	19845367	G	A	rs6426833	regulatory_region_variant	0.8324	gwas
10	GCST003043	Immune system	1	19845367	G	A	rs6426833	regulatory_region_variant	0.4676	gwas
11	GCST005527	Immune system	3	48468811	C	T	rs3135952	3_prime_UTR_variant	1.0000	gwas
12	GCST000964	Immune system	11	94193291	T	G	rs16920014	upstream_gene_variant	1.0000	gwas

3. SUMMARY AND DISCUSSIONS

In summary, 1) HDFS and Hive/Impala is good for storing big data in the cloud. The raw data can be stored in S3 with infinite space. We can take advantage of the schema on read providing the most flexibility, however, the platform lacks some analytic functions. 2) For PostgreSQL, it is a little time-consuming for data preprocessing and designing schema in order to inject data into the database. However, the data query speed is the fastest among all database choices for the test datasets. 3) For TileDB, it has good concept on big data storage and slicing; native distributed data storage in cloud. However, during the implementation, I had met a lot of bugs. Considering the short history, it is reasonable that it is still pretty buggy at this stage, especially data type support and compatibility issues with Python and Spark. A big caveat I think is that the data slicing and feature filtering are optimized within the array. For the joined operation between arrays seem to be dependent on other framework, like Pandas/MariaDB/Spark. 4) For MongoDB, it is very easy to inject JSON and CSV data into database. But it was really a pain to perform join operations between collections and speed is very slow! 5) The final option using MongoDB + pySpark, which is the distributed data storage and big data analytics. SparkSQL can complement the aggregation and analyze function of MongoDB. It has a lot of potential for working on the complete Open Target Datasets.

Big data analytics face challenges at every stage of the pipeline, from data storage, data ETL, data analytics and visualization. There are also many tools at each stage for data engineers and analysts to choose. The database solutions discussed in this project are exploratory and preliminary, and may be serve as non-exclusive examples of database implementation for other type of datasets.

Open Targets Platform and Genetics Portal is still pretty new, but an innovative and evolving big data platform for severing life science community. The current data architecture has the multiple components including databases and big data tools that deserve dedicated full-time jobs to learn and improve. Here are the brief summary points I got from the job description and requirements for the Technical Project Lead (Open Targets) position posted online. I think it is a very good summary of the technical skills for the position as well as for the platform.

- Scala and Spark for ETL pipeline
- Data is injected into ElasticSearch and ClickHouse
- Data is served in the front end using GraphQL APIs using Play framework and consumed by React-based web interface
- All applications are deployed and served in Google Cloud Platform using Terraform
- Besides all above, all resulting datasets are made available to public through different channels using Google BigQuery

4. SOURCES

1) Open Targets: https://platform-docs.opentargets.org/
2) A crash course in open targets: https://blog.opentargets.org/summer-school-crash-course-part-1/
3) A crash course in open targets: https://blog.opentargets.org/summer-school-crash-course-part-2/
4) A crash course in open targets: https://blog.opentargets.org/summer-school-crash-course-part-3/
5) TileDB documentation: https://docs.tiledb.com/main/
6) pySpark SQL documentation: http://spark.apache.org/docs/latest/api/python/getting_started/quickstart_df.html
7) pyMongo: https://pymongo.readthedocs.io/en/stable/tutorial.html