Lab Report Week 4 and Week 5

Lihui Lu
5/9/2017

Introduction

Last week we learned and practiced the usage of MySQL database to store and analysis biological data. We have practiced how to create tables in databases, parse information that we want from data acquired through other online biological databases, and pass the data into the database. Previously, although not much experiencing working with the databases, I have taken the basic database introduction course.

We have also practiced the usage of command line blast. We have learned how to build a blast database with a specific database and query another genome using that database.

MY SQL Table Implementation

We selected several characteristics of genomes that we want to store and will be using for later analysis. We have divided the information into separate genres and designed individual tables to store them. The idea of using multiple tables instead of a giant table is to reduce the amount of redundancy so that maintenance of the database will be easier in the future. We have created tables for genomes, replicons(some genome could contain multiple replicons, genes, exons, functions, gene synonyms, external references, and functions. The attributes in each table are described as below.

Note: previously auto-increment was used for genome_id, replicon_id and gene_id. However, gap of the auto-incremented id due to auto_increment_lock_mode option of the InnoDB was found. Thus, these ids were then generated manually by using counters during data insertion.

Table attributes and explanation

1. The genomes table

The genomes table	Explanation
genome_id	Unique id assigned to each genome, primary key
tax_id	Taxon id acquired from the genbank file
domain	Defines the genome as one of 'bacteria','archaea','eukarya'
genome_short_name	Abbreviation of the genome name
genome_long_name	Full name of the genome
size	Genome size in bp
release date	Date of release of the genome stored in DB

The attribute of accession number was in the genomes table but was later moved to the replicons table since we have noticed that for genomes with multiple replicons, each replicon has a different accession number.

2. The replicons table

The replicons table	Explanation
replicon_id	Unique id assigned to each replicon, primary key
genome_id	Refer to the genome that the replicon belongs to
name	Replicon name
num_genes	Number of CDS in the replicon(including pseudo genes
rep_size	Replicon size in bp
accession	Accession number for the replicon
type	Replicon type (either chromosome, plasmid or unknown if not specified in the genbank file)
structure	Replicon structure(either linear or circular)

3. Genes table

The genes table	Explanation
gene_id	Unique id assigned to each gene, primary key
accession	Refer to the genome accession in NCBI
genome_id	Genome that the gene belongs to
replicon_id	Replicon that the gene belongs to
locus_tag	Locus tag
name	name of the gene
strand	Either the gene is on '+' forward strand or '-' reverse strand
num_exons	Number of exons this gene is made up of
size	Size of gene in bp
product	Protein product of the gene

4. Exons table
   Since each gene can be made up of multiple exons, each gene_id would correspond to multiple exon start and end position.

The exons table	Explanation
gene_id	The gene that the exon belongs to
left_position	start position of the exon
right_position	end position of the exon
size	length of exon in bp

5. Gene_synonms table

   The gene_synonms table is also implemented to be storing the one-to-many relationship since each gene could have multiple synonyms.

The functions table	Explanation
gene_id	The gene that the synonyms belongs to

synonym	synonym of the gene

6. Functions table

The functions table	Explanation
gene_id	The gene that the function belongs to
function	function of the gene

Since each gene could have multiple functions, this should be implemented to be a one to many relationship in the future. For now, I have concatenated all the functions of one gene using ';' and stored them as one entity. Keeping a separate table storing all the functions and their identifications can help facilitate searching and querying gene functions in the future.

1. List all the tables in current database

```
SHOW TABLES;
```

2. Create the genomes table

```
CREATE TABLE genomes (
    genome_id   INT     (10) UNSIGNED NOT NULL AUTO_INCREMENT,
    tax_id      INT     (10) UNSIGNED NOT NULL,
    domain      ENUM('bacteria','archaea','eukarya') NOT NULL,
    genome_short_name VARCHAR (100) NOT NULL,
    genome_long_name VARCHAR (100) NOT NULL,
    size            INT     (10) UNSIGNED NOT NULL,
    release_date VARCHAR (100) NOT NULL,
    PRIMARY KEY (genome_id),
    KEY (tax_id)
)ENGINE=InnoDB;
```

3. Show how the table is created

```
SHOW CREATE TABLE genomes;
```

4. Show attributes of a table

```
DESCRIBE genomes;
EXPLAIN genomes;
SHOW COLUMNS IN genomes;
SHOW COLUMNS FROM genomes;
```

Loading Tables with Data

Different approaches can be taken in order to load tables with data, such as, using mySQL command, using the linux command line, and also, linking python directly to the database to do data manipulation within python scripts. After trying a couple of times, I have chosen to use the `pymysql` package and manipulate the database within the python script.

Some examples of data import:

1. Within mysql

```
#import data line by line
INSERT INTO table(attr1, attr2, attr3,...)
    VALUES(v1,v2,v3);

#import large amount of data at one time
LOAD DATA LOCAL INFILE 'in_file' INTO TABLE table
(attr1, attr2, attr2,.....);
```

2. Using Linux command line

```
mysqlimport -u user -p --local bimm185 --columns=attr1,attr2,.... in_file
```

3. Using pymysql

```
import pymysql

#DB parameters setup
hostname = 'localhost'
username = 'username'
password = 'password'
database = 'bimm185'
```

```
#Set up DB connection
myConnection = pymysql.connect(host=hostname, user=username, passwd=passwor
d, db=database, local_infile=True, autocommit=True)

#Query performance
cur = myConnection.cursor()
sql_statement = ("LOAD DATA LOCAL INFILE in_file INTO TABLE table"
    "(attr1, attr2, attr2.....);")
cur.execute(sql_statement)
cur.close()
```

When loading the data into the database, I have tried 2 different stratgies. At first, all the unique identifiers in all tables are set to be auto-increment so that the database can assign unique id to each entry without we manully setting the id up. However, the first attempt failed and the 2nd attempt was made where I manuly assign ids to each entry by using a counter.

1. 1st-attempt: using auto-increment to generate id automatically

Using the auto-increment setting in `InnoDB` would cause problem when I tried to append new data into the database instead of wiping out the whole database and load everything in. Since the default value of the `--innodb-autoinc-lock-mode` is set to 1, ids will be skipped if when trying to load them into the database and an error occured. For example, when I tried to load the genomes into the table again, the genomes table previous had genomes with id 1 and 2, but the 2 new genomes will have id 4 and 3. genome id of 3 will be skipped, regardless of which method I choose to load the data. I have tried to set the `--innodb-autoinc-lock-mode` parameter to 0 but didn't figure out how to do it.

Some explanations on different values for `--innodb-autoinc-lock-mode`:

- `innodb_autoinc_lock_mode`

Command-Line Format	--innodb-autoinc-lock-mode=#	
System Variable	Name	innodb_autoinc_lock_mode
	Variable Scope	Global
	Dynamic Variable	No
Permitted Values	Type	integer
	Default	1
	Valid Values	0
		1
		2

The lock mode to use for generating auto-increment values. Permissible values are 0, 1, or 2, for “traditional”, “consecutive”, or “interleaved”, respectively. The default setting is 1 (“consecutive”). For the characteristics of each lock mode, see InnoDB AUTO_INCREMENT Lock Modes.

1. 2nd-attempt: manually generate id using counter

   As a result, I have changed my approach and use counters each time when loading new data into the tables. First, queries were performed to get the max id that has been assigned to an entry in each table using `pymysql` (script see below). Then the base of the counter is set to be that number + 1. 3 counters were set, genome_counter, replicon_counter and gene_counter.

   Examples of querying the max id that has been assigned in the genome table:

```
def query_mx_genome_id(conn):
    cur = conn.cursor()
    cur.execute("SELECT max(genome_id) FROM genomes;")
    result = cur.fetchone()
    cur.close()

    if result[0] is None:
        #if the genomes table is empty
        return 0
    else:
        return result[0]
```

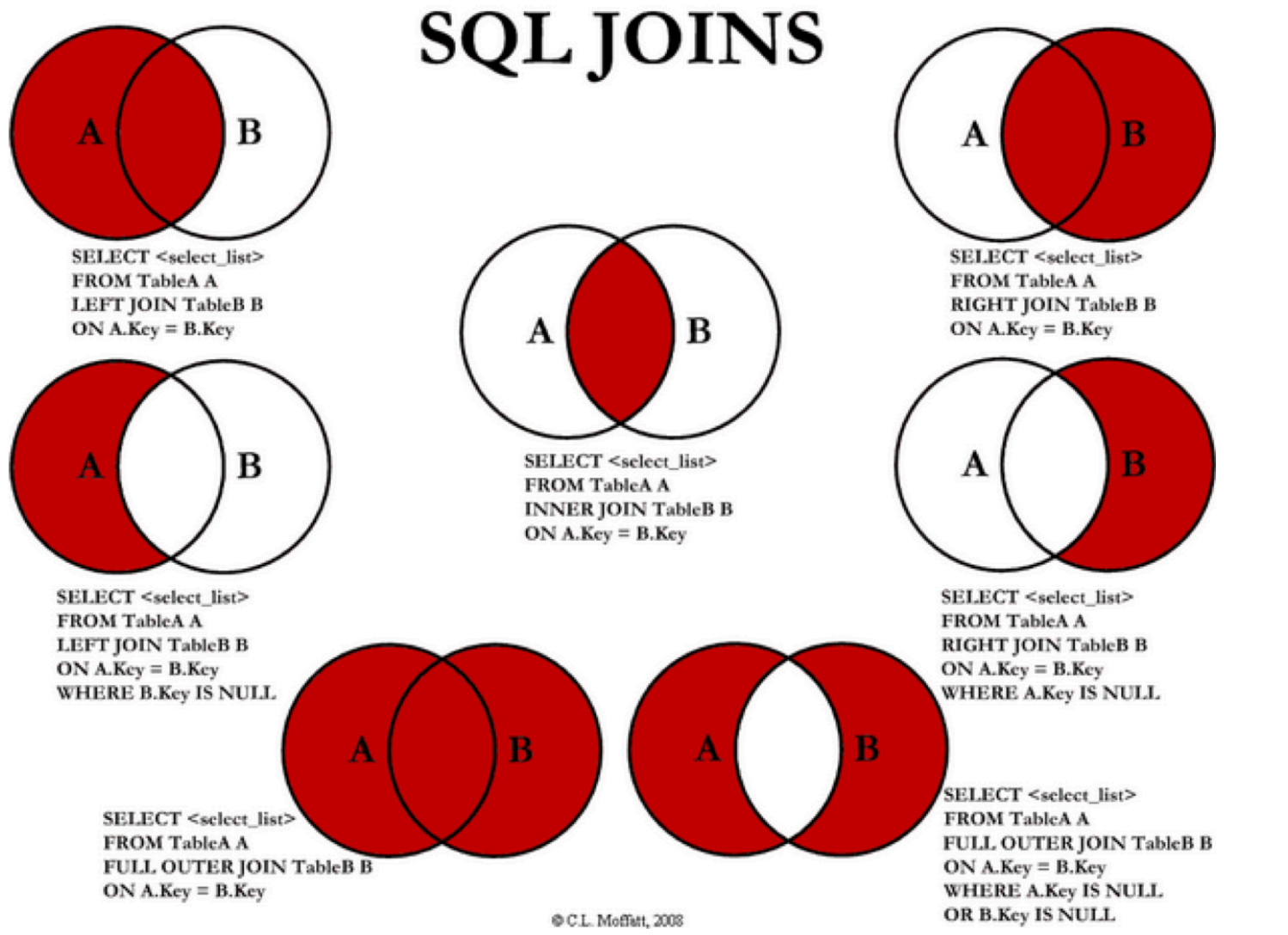
Querying Tables

Relational database allowed us to minimize the amount of redundant data that we need to store without loosing any information(if designed correctly) and query the desired result in a speedy fashion. Regular expressions can also be used in queries.

Some basic queries that we can do are:

```
SELECT attributes FROM table
WHERE condition 1 AND condition 2 .....;

SELECT d1.attributes, d2.attributes FROM table1 d1, table2 d2
WHERE condition 1 AND conditions 2 .....;

#get unique values of attribute
SELECT DISTINCT(attribute) FROM table;

#sort result
SELECT attributes FROM table ORDER BY attr1 DESC;
SELECT attributes FROM table ORDER BY attr1 ASC, attr2 DESC;

#group results and return count
SELECT count(*) FROM table GROUP BY attr1;

#group results and return max(attr2) for each group
SELECT attr1, max(attr2) FROM table GROUP BY attr1;
```

Joins can also be used to query tables using the relationships between tables. There are many type of joins in SQL. Joins can be divided into 2 large categories basically, inner join and outter join. Where inner join only includes entries that exists in both table, outter join allows the entry to be missing in on of the table. In short, inner join is intersection and outter join is union. A diagram can be used(borrowed from lecture slide) to explain the join relationship in SQL.



Other advanced queries such as recursion can also be used in relational database to solve problems when necessary.

Editing Tables

Here you could generate subsection for deleting tables, deleting columns, deleting rows, renames a column, renaming a table, modify the definition of a column, copying data from one table to another, etc.

Tables in the DB can also be edited later after they are created when we found a better implementation of the previous table.

Some examples of table editing:

1. Delete a table and all its contents:

```
DROP TABLE genomes;
DROP TABLE IF EXISTS genomes; <- safer choice to delete a table with checking
```

1. Delete all the rows within a table but kept the table

```
TRUNCATE genomes; <- resets AUTO_INCREMENT counter
DELETE FROM GENOMES; <- clears rows but does not reset AUTO_INCREMENT counter
```

1. Delete a column

```
ALTER TABLE table DROP attribute;
```

1. Insert a new column

```
ALTER TABLE table ADD attribute type;
```

1. Rename A column

```
ALTER TABLE table CHANGE attribute1 attribute2 type;
```

1. Change the type of a column

```
ALTER TABLE table MODIFY attribute new_type;
```

1. Modify the value of a field in a table

```
UPDATE table SET attribute1 = v1 WHERE attribute2 = v2;
```

Homology and BLAST

During week 5, we have learned the definition of homology and practiced using Blast to get

homologous gene from two different genome. There are 3 types of homology that we learned, paralogy, where genome sequences diverge after gene duplication, orthology, where genome sequences diverge after speciation, and xnelogy, where a foreign piece of genome is introduced.

Blast

Commands that can be used to create a blast database by providing the protein sequence

```
zcat < GCF_000005845.2_ASM584v2_protein.faa.gz | makeblastdb -input_type 'fasta
' -dbtype prot -parse_seqids -hash_index -out /Users/LunaLu/Dropbox/Documents/S
P17/BIMM185/blastdb/E_coli -title "ecoli  04/27/2017"  -in -
```

Blast one protein sequence against another

```
zcat < ../Week4/genome/A.tumefaciens/*.faa.gz | blastp -query - -out A.tumefaci
ens_vs_E_coli_k12.out -db $BLASTDB/E_coli -evalue 0.01 -outfmt '6 qseqid sseqid
 qlen slen bitscore evalue pident nident length qcovs qstart qend sstart send'
```

2 blast were performed. A.tumefaciens was blasted against E.coli and E.coli was blasted against A.tumefaciens. The blast result can then be imported into the mySQL database. Queries can be used to determine homology between proteins.

A query was performed to identify orthology using the Bi-directional Best Hit definition of orthology. For each query sequence, the subject sequence with the highest bitscore was first picked out for both blast tables. Then they were inner joined together where the query sequence of the first blast should be the subject sequence of the second blast. At last, the results are sorted in descending order by bitscore.

```
select b1max.qseqid, b1max.sseqid, b1max.bitscore, b1max.qcovs, b1max.scov, b2m
ax.bitscore, b2max.qcovs, b2max.scov from
(select b.qseqid, b.sseqid, b.bitscore, b.qcovs, b.scov from blast_1 b,
    (select b1.qseqid,max(bitscore) as bitscore from blast_1 b1 group by b1.qse
qid) maxtable
    where b.qseqid = maxtable.qseqid and b.bitscore = maxtable.bitscore) b1max,

(select b.qseqid, b.sseqid, b.bitscore, b.qcovs, b.scov from blast_2 b,
    (select b2.qseqid,max(bitscore) as bitscore from blast_2 b2 group by b2.qse
qid) maxtable
```

```
    where b.qseqid = maxtable.qseqid and b.bitscore = maxtable.bitscore) b2max
where b1max.qseqid = b2max.sseqid and b1max.sseqid = b2max.qseqid
order by b1max.bitscore DESC;
```

The highest bitscore was 1660 while the lowest bitscore in the result set is 28.9. A total of 1622 pairs of genes were defined. The result of the BDBH is plotted below. It can be observed that for higher bit scores, both of the query coverage and the subject coverage tend to coverge to 100. And when the qcov/scov value was plotted over the bitscore, it can be observed more clearly that not only the coverage itself coverage to 100, the both coverage values tend to be closer to each other. The ratio converges to 1 as the bitscore goes up. This result aligns with our expectation that the higher the score is, the two sequences are more similar to each other and thus their alignment length would increase.

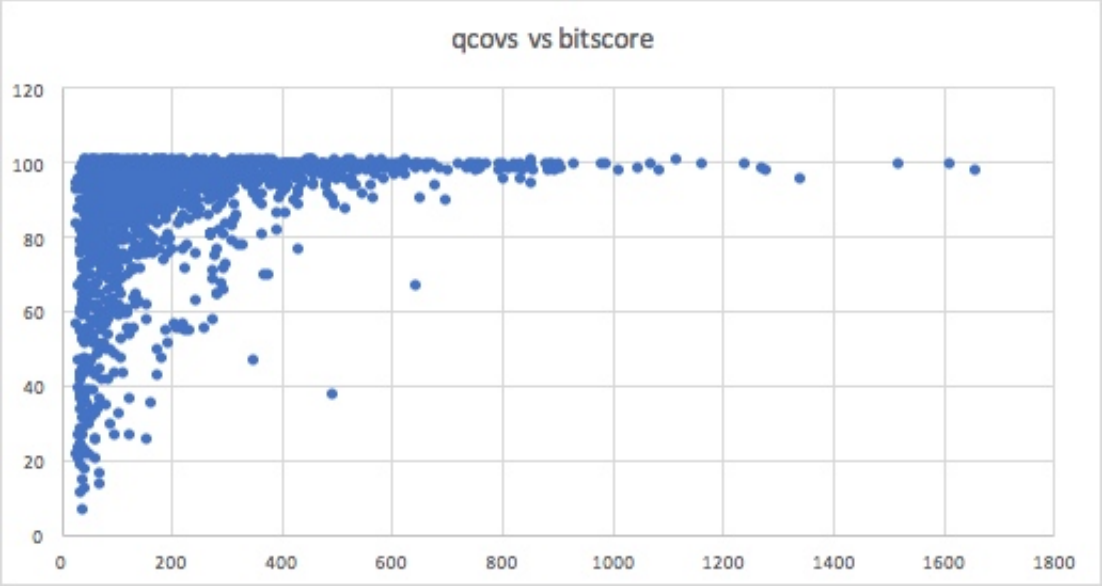


Figure 1. Query coverage vs blast bitscore resulted by blast E.coli against A.tumefaciens

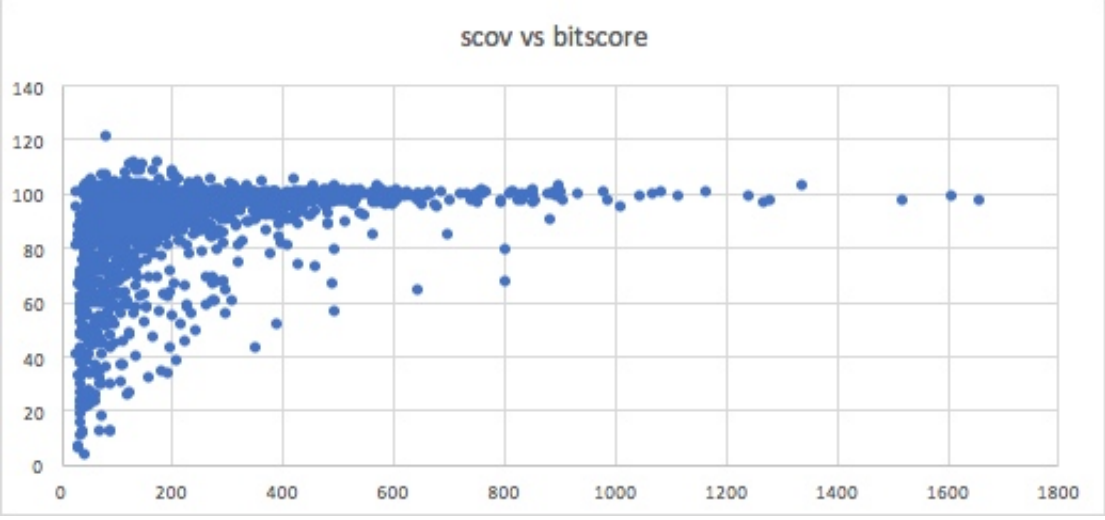


Figure 2. Subject coverage vs blast bitscore resulted by blast E.coli against A.tumefaciens

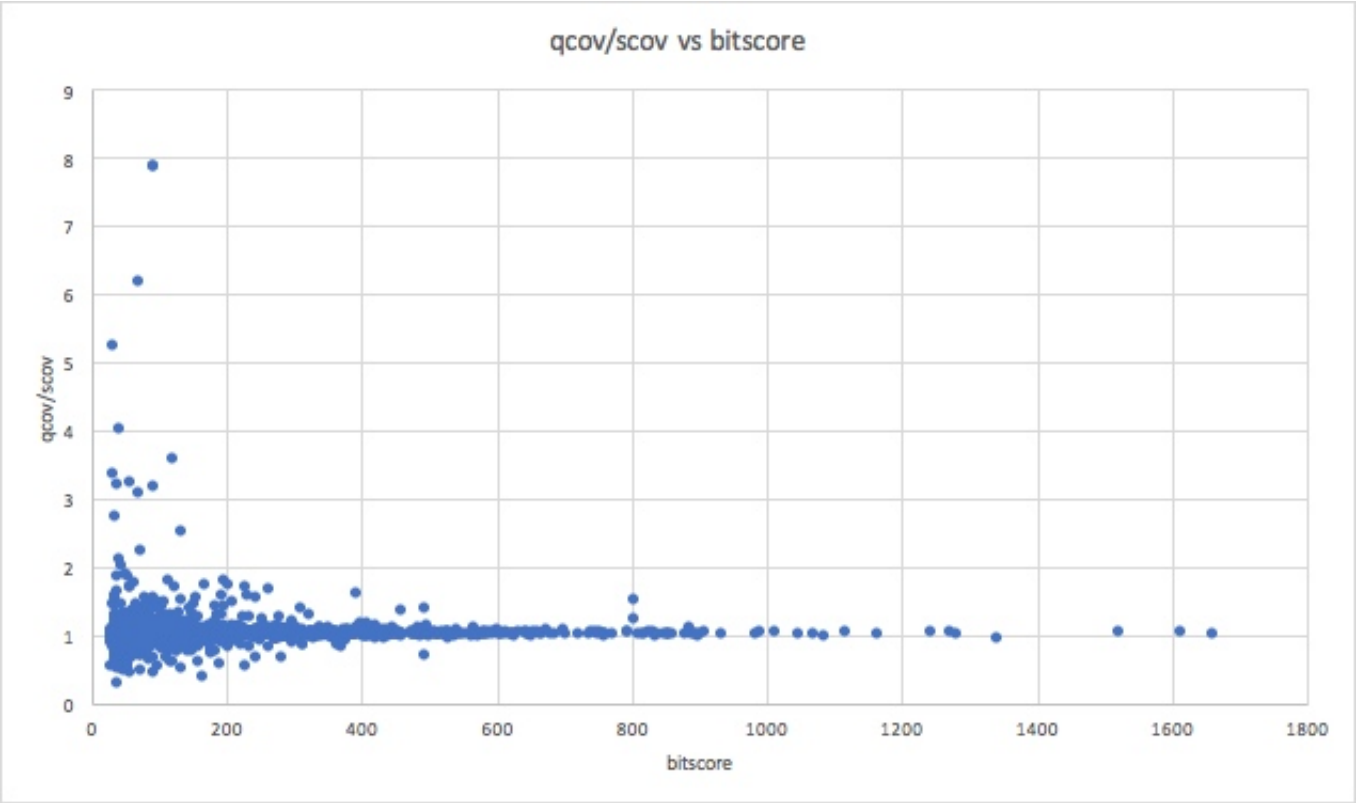


Figure 3. Query coverage/Subject coverage vs blast bitscore resulted by blast E.coli against A.tumefaciens

Discussion

The past two weeks we have learned how to import data into the relational database and use the relational database to solve some questions in a faster fashion. While the design of a relational database can affect the ease of usage and maintanance in the future, it's important to keep each table as simple as possible the prevent duplicate information so that when we need to update or alter some information in the database, it can be easier and don't need to consider too many edge cases.