Slightly more sophisticated SQL

Today we’re going to work a little bit more with SQL. Interspersed amongst the direction to follow are numbered paragraphs. Each numbered paragraph will correspond to file or table that will be part of your graded homework. The bolded words in these paragraphs refer to what I will check to determine your grade. The numbered paragraphs will require you to do a little bit of thinking on your own in order to perform the appropriate task. In several cases that task will not be complete without extra output/manipulations from non-numbered paragraphs. Ask me if anything is unclear.

In the home directory for student is a file called samples.txt. It is in FASTA format.

1. Create a table named ‘samples’ with two columns. One column should be named ‘id’ and the other ‘seq’. The column ‘id’ should hold 10 characters, and ‘seq’ should hold 20. Neither should allow NULLS (hint… use NOT NULL). When creating the table, make id the primary key, and make an index called ‘seq’ that is an index for ‘seq’.

2. Now load data into the table from the file samples.txt. You’ll want to use the ‘LOAD DATA’ command. There are a few things to be careful about:
   • INFILE might not work. Consider using LOCAL INFILE
   • The directory in which the mysql session was started will affect where the program looks for files. The filename ‘samples.txt’ might not work for. You may need to use ‘/home/student/samples.txt’.
   • The file samples.txt is not tab delimited. But you can think of it as being ‘\n’ delimited. Play around with the TERMINATED BY clause.

If you’re done things correctly then you should have a table named ‘samples’ with 1000 records. Check the first 5 like this:

```sql
mysql> SELECT * FROM samples LIMIT 5;
+----+----------------------+
| id | seq                   |
|----+----------------------+
| >1 | TGTATGTATTATTATTGTTC |
| >2 | AGGAGGTCGGCCACTAATAG |
| >3 | TCGTGCCAGCCACCACGGTC |
| >4 | TTTAGGGCTGGGCACTAGTG |
| >5 | CCAACACTCAATAAGGCACT |
```  

Use OFFSET to check a few records near the bottom:

```sql
mysql> SELECT * FROM samples LIMIT 4 OFFSET 900;
+----+----------------------+
| id | seq                   |
|----+----------------------+
| >901 | GGTTTCTAAGACCAATGGAT |
| >902 | AGTGCTACTGTTAAAGAGT |
| >903 | CTTCACACAGGATTTGTC |
| >904 | CAGGTCTAACAAGACCTTAC |
```
Now, the column ‘id’ is unique and does its job, but the caret ‘>’ at the beginning, offends my sense of aesthetics, so let’s get rid of it. Before doing the update though, let’s figure out what needs to be done. I think that ‘substr(id,2)’ should strip away the left-most character from id, but I’d like to be certain so I’ll do a query first:

```
SELECT substr(id,2) AS id, seq FROM samples LIMIT 5;
```

That looks good, but I’d better check near the end of the table too—just in case I’ve misunderstood/misremembered the meaning of the arguments passed to ‘substr’, or in case there is a side-effect involving the length of ‘id’ that I have overlooked (If I were a better person I’d randomly select several locations from throughout the ‘samples’ table… but this is easy to understand and that seems to be unnecessary work.)

```
SELECT substr(id,2), seq FROM samples LIMIT 5 OFFSET 900;
```

Ok… everything looks good. Now let’s use the UPDATE command to fix the table:

```
UPDATE samples SET id = substr(id,2);
```

Let’s look at the samples along with their reverse complement:

```
SELECT id, lower(reverse(seg)) AS rc, seq FROM samples LIMIT 5;
```

(There are far better ways of doing the reverse complement than the one that we’re using here. For example you could define a function rc:

```
CREATE FUNCTION rc (str CHAR(20))
RETURNS CHAR(20) DETERMINISTIC
RETURN upper(
    replace(
        replace(
            replace(reverse(str), "A", "t"),
            "T", "a"),
            "C", "g"),
        "G", "c")
) AS rc,
seq
FROM samples LIMIT 5;
```

SELECT ID, rc(seq) FROM samples LIMIT 5;
3. Modify the query above so that the column ‘rc’ is upper case (instead of lower case) and, using the CREATE TABLE, output lines 100-200 of the query into a table called samples_f_r (You may find it useful to modify the LIMIT OFFSET clause in the query).

Add a column named ‘pos’ of type INT using ALTER TABLE. Using UPDATE and RAND() add a random number between 1 and 20 into each entry of the ‘pos’ column. The function ‘rand()’ generates a number greater than or equal to 0, but less than 1. Since the column ‘pos’ is an integer mySQL will round for you, but this means you need to take extra care to ensure that the random numbers that you generate are evenly distributed—in other words, you need to be certain that, for example, a 20 is as likely to be produced as a 1. Since don’t want to depend upon mysql’s rounding—perhaps in future versions they’ll decide to truncate instead of rounding— an expression such as 1+floor(20*rand()) is a good one to use.

Add a column of type CHAR(1) named ‘strand’ to ‘samples_f_r’. We are going to use ‘strand’ to simulate the existence of forward and reverse complement matches, so I want about half the entries to be ‘+’ and about half the entries to be ‘-’. In order to do this, we’re going to need the ‘case’ statement:

```
CASE WHEN rand() < .5 THEN '+' ELSE '-' END
```

After you’ve added the ‘strand’ column, use this command to randomly fill it with ‘+’ and ‘-’:

```
UPDATE samples_f_r
SET strand = CASE WHEN rand() < .5 THEN '+' ELSE '-' END;
```

Add the column ‘old’ of type CHAR(1) to ‘samples_f_r’. The last action you need to perform on ‘samples_f_r’ is to fill the ‘old’ column: For each record, if strand is ‘+’ then change ‘old’ to be the nucleotide at ‘pos’ from ‘seq’. If strand is ‘-’ then change ‘old’ to be the nucleotide at ‘pos’ from ‘rc’. You may find the SUBSTR function useful.

4. Create a directory in /tmp with your first name. Now write a query that will return ‘id’ followed by ‘seq’ (if ‘strand’ is +) or ‘rc’ (if ‘strand’ is -)use the clause INTO OUTFILE to output the query to /tmp/<name>/test.txt

Some more queries and the affect of indices

Perhaps we’re interested in how many times specific samples are repeated. Here is one way we could find all the samples that are repeated:

```
SELECT a.id,b.id,a.seq FROM samples AS a, samples AS b WHERE a.seq=b.seq AND a.id != b.id;
```

But there’s a better way:

```
SELECT count(id) AS count,seq FROM samples GROUP BY seq HAVING count > 1;
```
That was all pretty fast! Let’s found out what would happen if we did NOT have indices. Let’s start by disabling them:

```
ALTER TABLE samples DISABLE KEYS;
```

Now let’s reset the query cache (just to make certain that our earlier query isn’t influencing our results) and see how the first example fairs:

```
RESET QUERY CACHE
SELECT a.id, b.id, a.seq FROM samples AS a, samples AS b WHERE a.seq=b.seq AND a.id != b.id;
```

Quite the difference isn’t it? Feel free to try the second example – it does MUCH better. In any event, let’s re-enable the indices:

```
ALTER TABLE samples ENABLE KEYS;
```

Design and create a table to hold the data from the file ‘align.txt’ that you made during the previous assignment (you won’t be graded on this here, since this step is part of the final). I’m going to assume that your table has columns named ‘id’, ‘loc’, ‘strand’, and ‘seq’. It should have others as well, but I’m assuming that ‘id’ is the name of your sequence, ‘loc’ is the location of the alignment, ‘strand’ is ‘+’ if the read aligned against the forward strand, and it is ‘-’ if the read aligned against the reverse strand. The value of ‘strand’ affects the contents of ‘seq’. If ‘strand’ = ‘+’, then ‘seq’ is identical to the 20 nucleotide sample in ‘samples.txt’ with id ‘id’ (also from last homework). If ‘strand’=‘-’, then ‘seq’ is the reverse complement. As a consequence, ‘seq’ is no more than a few characters different from some substring in the reference genome—no matter the value of ‘strand’. And we can use that to write queries that make it easy to see all the covering reads for a fixed location.

To start, let’s find out how which reads covered location 10000:

```
SELECT * FROM tmp WHERE loc<=10000 AND loc >= (10000-19);
```

Why did I choose 10000-19? Let’s look at the alignment in a better way. Notice that the left-most read that can cover location 10,000 starts at location 10,000-19. We need to add enough spaces to a covering sequence so that in a mono-spaced font, the left-most location of each padded string corresponds to location 10,000-19:

```
SELECT lpad(seq, loc-(10000-19)+20,"") AS seq, strand, loc FROM tmp WHERE loc <=10000 AND loc >= (10000-19);
```

Let’s just add a row at the bottom and the top that has an asterisks denoting the desired location:

```
SELECT lpad("*",20,""),"",0
UNION ALL SELECT lpad("*",20,""),"",0;
UNION ALL SELECT lpad("*",20,""),"",0;
```

We add the ALL keyword so that both rows with the ‘*’ appear. And I think we’ll call that good for this lab. Thanks!