Name: _____

Please answer the following questions within the space provided on the following pages. Should you need more space, you can use scratch paper, but clearly label on the scratch paper what problem it corresponds to. While you are not required to explain your queries, comments may help me to understand what you were trying to do and thus increase the likelihood of partial credit should something go wrong. If you get entirely stuck somewhere, explain in words as much as possible what you would try.

This is a pen and paper exam, and thus computers and internet capable devices are prohibited. If you have any confusion about question intention or wording, please do not hesitate to ask!

Your work must be your own on this exam, and under no conditions should you discuss the exam or ask questions to anyone but myself. Failure to abide by these rules will be considered a breach of Willamette's Honor Code and will result in penalties as set forth by Willamette's academic honesty policy.

Please sign and date the below lines to indicate that you have read and understand these instructions and agree to abide by them. *Failure to abide by the rules will result in a 0 on the test.* Good luck! You got this!

_____ _____

Signature Date

Question:	1	2	3	4	5	6	Total
Points:	12	10	12	18	18	0	70
Score:							

1. Below you are given the queries that created three tables and a view of what is currently in those tables.

tab1

```
CREATE TABLE tab1 (
  A TEXT,
  B INT CHECK (B > 0),
  C NUMERIC(4,2) UNIQUE,
  PRIMARY KEY (A, B)
);
```

A	B	C
Henry	2	4.23
Jake	1	10.52
Katy	3	-4.83
Wyatt	4	83.10

tab2

```
CREATE TABLE tab2 (
  D DATE PRIMARY KEY,
  E NUMERIC(4,2)
    REFERENCES tab1 (C)
);
```

D	E
2022-02-14	10.52
2022-01-30	83.10
2022-04-24	4.23

tab3

```
CREATE TABLE tab3 (
  F INT PRIMARY KEY,
  G REAL,
  H TEXT,
  I DATE
    REFERENCES tab2 (D)
  CHECK (F > G)
);
```

F	G	H	I
2	1.1	Jake	2022-01-30
8	6.445	Katy	2022-04-24
0	-0.24	Henry	2022-01-30

Using this information, for each of the following queries, determine whether that query would run successfully or not. If it does not, explain directly what error would occur / why it occured.

(2) (a)
```
INSERT INTO tab3 VALUES
(7, -4, 'Katy', '2022-02-14');
```

Solution: This should be fine

(2) (b)
```
ALTER TABLE tab3 ADD FOREIGN KEY (H) REFERENCES tab1 (A);
```

Solution: This will error, since column A in tab1 is not necessarily a unique column *by itself*. That is, there could be multiple rows in column A with the same name, which would confuse the foreign key matching.

(2) (c)
```
UPDATE tab2
SET E = -4.83;
```

Solution: This should be fine, as -4.83 is a valid entry in the referencing tab1.

(2) (d)
```
DELETE FROM tab1
WHERE A = 'Katy';
```

Solution: This will be ok, because nothing is currently referencing the -4.83 that corresponds to the Katy row.

(2) (e)
```
ALTER TABLE tab3 ALTER COLUMN G SET DATA TYPE INT;
```

Solution: This will cause an error, as the -0.24 will be rounded to 0 at which point it breaks the $F > G$ constraint.

(2) (f)
```
ALTER TABLE tab1 ADD PRIMARY KEY (C);
```

Solution: This will error, as tab1 already has a primary key defined and you can not have more than 1 primary key.

2. Suppose you have the simple table below (named `regexes`) which contains only a single column of text values:

value
text
'APR 24 2022 12:23:42 -- 3321-2635-2673-3481 -- Hazel Western paid $50.23'
'MAR 18 2022 04:12:56 -- 8352-1634-8492-1642 -- Bob Johanssen paid $100.56'
'JAN 31 2022 15:02:14 -- 0234-1004-2835-1666 -- Karen Hinter paid $19.20'

For each of the below regular expression matching queries, determine what the output table would look like.

(2) (a)
```
SELECT (regexp_match(value, '-(\d{4})'))[1] FROM regexes;
```

A.	B.	C.	**D.**
3481	2022	3321	**2635**
1642	2022	8352	**1634**
1666	2022	0234	**1004**

(2) (b)
```
SELECT (regexp_match(value, '.*(\d)\.'))[1] FROM regexes;
```

A.	**B.**	C.	D.
3	**0**	2	3
6	**0**	1	8
0	**9**	3	0

(2) (c)
```
SELECT (regexp_match(value, '[A-Z][a-z]*'))[1] FROM regexes;
```

A.	B.	C.	D.
A	H	APR	Hazel
M	B	MAR	Bob
J	K	JAN	Karen

(2) (d)
```
SELECT (regexp_match(value, '[A-Z][a-z]+'))[1] FROM regexes;
```

A.	B.	C.	**D.**
A	H	APR	**Hazel**
M	B	MAR	**Bob**
J	K	JAN	**Karen**

(2) (e)
```
SELECT (regexp_match(value, '\d{3}[5-9]'))[1] FROM regexes;
```

A.	B.	C.	D.
2635	NULL	2022	3321
NULL	8352	2022	8352
2835	1666	2022	0234

(12) 3. Suppose you have a table named `enigma`, for which you know nothing of its contents. However, when you run the three queries below, you get the shown output. Determine a possible schema *and contents* for `enigma` that would satisfy all the below queries. **There is not a unique answer here. But your potential table needs to be consistent with the queries and outputs.** Make sure to indicate your column names and types as well as the actual row contents.

Query 1

```
SELECT
  COUNT(*),
  SUM(c1)
FROM enigma;
```

Output 1

count	sum
5	25

Query 2

```
SELECT c1
FROM enigma
WHERE to_tsvector(c2) @@
  to_tsquery(
    '(raccoon | swirl) & !clean'
  );
```

Output 2

c1
4
5
7

Query 3

```
SELECT (
  SELECT COUNT(*)
  FROM REGEXP_SPLIT_TO_TABLE(c2, ' ')
  )
FROM enigma
WHERE c1 = (
  SELECT COUNT(*)
  FROM REGEXP_SPLIT_TO_TABLE(c2, ' ')
  );
```

Output 3

count
4
5
7
3
6

I've included a full blank page on the next page you can use for scratch-work if you need. Just make sure it is clear to me where your final table is written down.

Solution:

- Query 1 tells us that there are 5 rows in the table and that summing up the `c1` column gives a value of 25, which presumably means that `c1` must be some sort of number column
- Query 2 tells us that we can do full text searches on the `c2` column, which implies that it must be a text based column. When we search the column for entries that contain "raccoon" or "swirl" but not "clean", we get the rows that have the `c1` values of 4,5, and 7. Which tells us both what must be in `c1` and further reinforces the idea that `c1` seems to be some sort of integer.
- Query 3 tells us that the value that is in `c1` is equal to the number of words in `c2`. We know this because, after filtering by that condition, we get out 5 rows, which is all of them. And they all add up to 25, so we have that confirmed as well.

So at that point we have the contents of `c1`. All that remains is populating `c2` so that the terms raccoon or swirl show up on the necessary lines and so that each line has the correct number of words. One final option might look like:

c1 *int*	**c2** *text*
4	`'The raccoon eats fish'`
5	`'It swirl it in water'`
7	`'Raccoons like to be up at night'`
3	`'Rats are bad'`
6	`'Rats require cleaning often and stink'`

4. Many GPS tracking units are capable of outputting tabulated data, which frequently involves a simple timestamp alongside the determined latitude and longitude measurements. Suppose you have such a table, which also includes a serial column to uniquely determine the ordering of the points, as seen below:

gps_tracks	
id	*serial*
time	*timestamp*
latitude	*double precision*
longitude	*double precision*

Here you can safely assume that the serial `id` increases in steps of 1 with no gaps (which isn't always guaranteed for serial columns). Use this information to write queries to answer the following questions. You can assume that any necessary extensions have already been added to the database. All of these could be done as a single query, but you can use multiple queries if you prefer (though nothing that would require a user to read a value off one table and enter it into a different query).

(6) (a) What is the straight-line distance between the initial point in the table and the final point? Note that this is not the actual distance traveled, but just the distance directly between the first and last point.

Solution: One possible solution might look like:

```
WITH
gps_points (id, time, geom) AS (
  SELECT
    id,
    time,
    ST_SetSRID(ST_MakePoint(long, lat), 4326)
  FROM gps_route
)
SELECT
  ST_Distance(
    (SELECT geom FROM gps_points WHERE id=1)::geography,
    (SELECT geom FROM gps_points WHERE id=(
      SELECT max(id) FROM gps_points))::geography
  );
```

(6) (b) What is the total actual distance traveled moving from point to point along the sequence of coordinates?

Solution: I found it easiest to use a CTE again here for the pre-processing:

```
WITH
gps_points (id, time, geom) AS (
  SELECT
    id,
    time,
    ST_SetSRID(ST_MakePoint(long, lat), 4326)
  FROM gps_route
),
adj_points (r_start, r_end) AS (
  SELECT g1.geom, g2.geom
  FROM gps_points as g1
  JOIN gps_points as g2
    ON g2.id = g1.id + 1
)

SELECT
  SUM (
    ST_Distance(
      r_start::geography,
      r_end::geography
    )
  )
FROM adj_points
;
```

(6) (c) Between which two ids was the speed the greatest? As a reminder, speed is a measure of distance over elapsed time.

Solution: This could use the same CTE as above, and just run a different final query

```
WITH
gps_points (id, time, geom) AS (
  SELECT
    id,
    time,
    ST_SetSRID(ST_MakePoint(long, lat), 4326)
  FROM gps_route
),
```

```
adj_points (id_start,
      id_end,
      t_start,
      t_end,
      r_start,
      r_end) AS (
  SELECT
    g1.id, g2.id, g1.time, g2.time, g1.geom, g2.geom
  FROM gps_points as g1
  JOIN gps_points as g2
    ON g2.id = g1.id + 1
)

SELECT
  id_start,
  id_end,
  ST_Distance(r_start::geography, r_end::geography) /
  (date_part('epoch', t_end) -
   date_part('epoch', t_start)
  )::real AS speed
FROM adj_points
ORDER BY speed DESC
LIMIT 1
;
```

5. You have the following table of information about western states in the continental US.

name *text*	**neighbor_count** *int*	**vote_color** *text*	**year_admitted** *int*	**num_reps** *int*	**avg_elev_ft** *int*
Oregon	4	blue	1859	5	3300
Washington	2	blue	1889	10	1700
Idaho	6	red	1890	2	5000
California	3	blue	1850	53	2900
Nevada	5	blue	1864	4	5500
Montana	4	red	1889	1	3400
Wyoming	6	red	1890	1	6700
Utah	6	red	1896	4	6100
Colorado	6	blue	1876	7	6800
Arizona	4	red	1912	9	4100
New Mexico	4	blue	1912	3	5701

Using this table, determine the output of the following queries.

(6) (a)
```
SELECT
  SUM(neighbor_count) AS result
FROM states
WHERE avg_elev_ft > 3000
GROUP BY year_admitted
HAVING COUNT(*) > 1
ORDER BY year_admitted
;
```

Solution: This only returns two values after the filtering and having:

result
12
8

(6) (b)

```
WITH
heights (name, tier) AS (
  SELECT
  name,
  CASE
    WHEN avg_elev_ft < 3000 THEN 'Low'
    WHEN avg_elev_ft < 5000 THEN 'Mid'
    ELSE 'High'
  END
  FROM states
)

SELECT
  h.tier,
  vote_color,
  COUNT(*)
FROM states AS s
JOIN heights AS h
  ON s.name = h.name
GROUP BY h.tier, vote_color
HAVING SUM(num_reps) > 5
ORDER BY h.tier, vote_color
;
```

Solution: This outputs:

tier	vote_color	count
High	blue	3
High	red	3
Low	blue	2
Mid	red	2

(6) (c)

```
SELECT name
FROM states as s1
WHERE EXISTS (
  SELECT 1
  FROM states as s2
  WHERE s1.neighbor_count < s2.neighbor_count
  ) AND name !~ '^[N-Z]'
ORDER BY name
;
```

Solution: We just get 3 outputs here:

name
Arizona
California
Montana

(4 (bonus)) 6. Compare and contrast a relational database with a classic spreadsheet like Excel. Where does each excel in your opinion? What are the weaknesses of each?

Solution: A relational database uses many tables to store both information and connections between different tables. This generally has the result of being more efficient in storing information only a single time. A classic spreadsheet application does not have this ability to relate tables as easily, and thus information that is needed in a particular context frequently needs to be included within that table, possibly causing duplication of information across multiple spreadsheets.

The strength of relational databases lies in the fact that they are more efficient at storing information in this way, and because they have built in schemas that mandate that only data of certain types or forms can be stored within the tables. This helps ensure data consistency and cleanliness across the database. Moreover, because they have the concept of joining tables, it is straightforward to bring together potentially vast amounts of data from different tables to investigate a particular feature.

The weakness of relational databases is that schemas reduce flexibility, and can be a pain to change if the data that you want to store changes. Additionally, inserting, removing, or changing data from the tables is no longer as straightforward, and require knowledge of SQL.

This is the real strength of spreadsheets: they provide a supremely easy method for every day users to insert, change, or remove data from the table, as well as providing an easy, always-on visualization of the data in table form. They can also provide other nice ammenities like graphing or other data visualization.

The weakness of spreadsheets lies in their lack of data verification and in their cumbersome nature when individual tables start getting massive. They aren't really meant for large, bulk storage, and it shows when they grow particularly large.

Thanks for the great semester, and sticking with me as I taught this class for the first time! Hopefully you got a lot out of it, and it proves useful in the future awesome things I know you'll go on to do! Never hesitate to drop me a line if you think there is something I could help you with going forward in the future. And enjoy your summer!