Tufts University
CS 115: Database Systems
Problem Set 3
Spring 2022

Complete the following exercises to the best of your ability and submit your answers on Gradescope by 11:59 PM on Wednesday, March 30, 2022. There will be a 10% deduction for submissions made by Thursday, March 31 at 11:59 PM, and a 20% deduction for submissions made by Friday, April 1 at 11:59 PM.

You can access Gradescope here: https://www.gradescope.com/. Click Log In and then School Credentials and scroll down to Tufts to log in using your Tufts credentials.

Format your answers as a PDF file. I recommend making a copy of this file as a Google Doc (File > Make a copy), filling in your answers, and then saving it as a PDF (File > Download > PDF document). You can also download this Google Doc as a Word document and save that as a PDF, or use other software entirely.

If you choose to, you can work with 1-2 partners on this assignment. If you worked with partners, mark in Gradescope the partners that you worked with. Make only one submission per group.

Direct any questions about the assignment to Piazza.

Problem 1 (10 points)

Consider the following version of a `Movie` table:

```
Movie(id CHAR(5), name VARCHAR(20), year INTEGER, rating VARCHAR(5))
```

And consider the following tuple from that table:

```
('98148', 'Encanto', 2021, 'PG')
```

a. What would this tuple look like if we stored it in a *fixed-length* record? Put your answer in the table below. Follow the conventions used in the class notes, where each cell is one of:

- A data field, such as an entire string or integer
- A delimiter character
- A field length
- An offset

Make the width of each cell larger or smaller as needed, and observe the following conventions:

- Use a number sign (`'#'`) as a delimiter when it is necessary to record the end of a variable-length field's value.
- Use an underscore (`'_'`) to represent any bytes of wasted bytes (i.e., bytes that are part of the record's representation but are not actually storing useful data or metadata)
- You should not leave any empty cells between the components of the record. The only empty cells should be at the very end of the record, if you don't end up needing all cells.

98148	Encanto#____________	2021	PG#__						

b. What is the length of the record (in bytes) that you constructed in part (a)? Assume all characters are 1 byte long and all integers are 4 bytes long.

id → 5 bytes
name → 20 bytes
year → 4 bytes
rating → 5 bytes
In total, the length of the record is 34 bytes long.

c. What would this tuple look like if we stored it in a *variable-length* record with the format that begins with a header of offsets? Put your answer in the table below. Use the same conventions as in part (a).

20	25	32	36	38	98148	Encanto	2021	PG	

d. What is the length of the record (in bytes) that you constructed in part (c)? Use the same assumptions as stated in part (b).

offsets → 20 bytes (4 bytes each)
id → 5 bytes
name → 7 bytes
year → 4 bytes
rating → 2 bytes
In total, the length of the record is 38 bytes long.

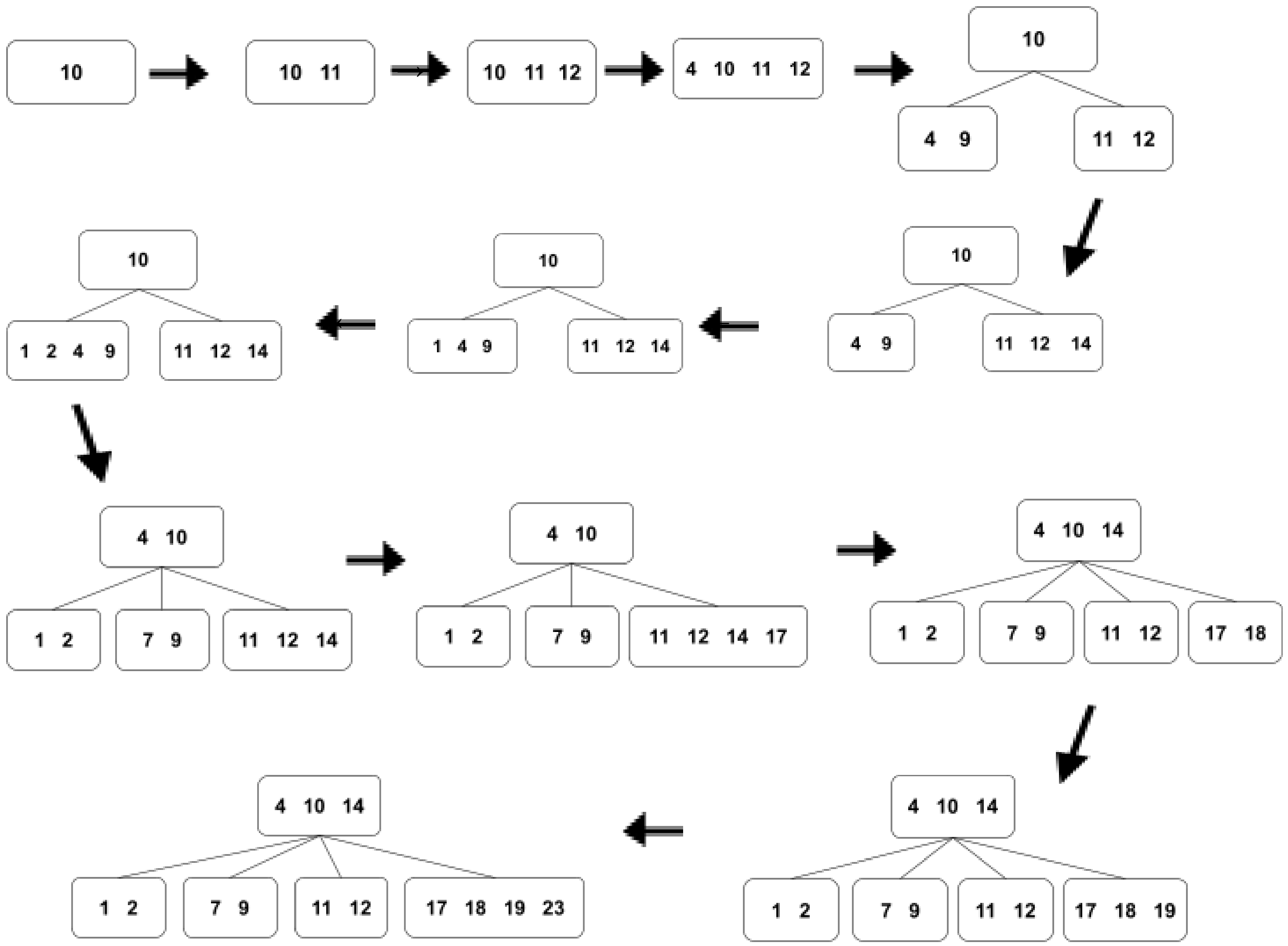
Problem 2 (10 points)

Say that you want to insert items with the following sequence of keys into a collection of records that uses some form of indexing:

10, 11, 12, 4, 9, 14, 1, 2, 7, 17, 18, 19, 23

a. Insert these keys into an initially empty B-tree of order *m*=2. As we did in class, show the tree after each insertion that causes a split of one or more nodes, and the final tree.

Below is an embedded Google Drawing that includes nodes of different sizes. ***Make copies of the diagram*** so that you can use separate diagrams for the results of each insertion that causes a split, and for the final tree. Note that you do **not** need to keep the shape of the tree that we have given you. Rather, you should edit it as needed: deleting or adding nodes and edges, replacing the Xs with keys, adding or removing keys, and making whatever other changes are needed.



Problem 3 (10 points)

Again consider the sequence of keys from Problem 2:

10, 11, 12, 4, 9, 14, 1, 2, 7, 17, 18, 19, 23

Insert this same key sequence into a hash table that uses linear hashing. The table should use the hash function *h(x)* = *x*, and it should start out with two empty buckets. Assume that a bucket is added whenever the number of items in the table exceeds **three times** the number of buckets ($f > 3n$). Use the tables provided to show the state of the table before and after each increase in the number of buckets, as well as the final state of the table.

before first increase

0	10, 12, 4, 14
1	11, 9

after first increase

0	12, 4
1	11, 9
2	10, 14

before second increase

0	12, 4
1	11, 9, 1, 7
2	10, 14, 2

after second increase

0	12, 4
1	9, 1
2	10, 14, 2
3	11, 7

before third increase

0	12, 4
1	9, 1, 17
2	10, 14, 2, 18
3	11, 7, 19

after third increase

0	
1	9, 1, 17
2	10, 14, 2, 18
3	11, 7, 19
4	12, 4

final state of the table

0	
1	9, 1, 17
2	10, 14, 2, 18
3	11, 7, 19, 23
4	12, 4

Problem 4 (15 points)

It's often advantageous to have a user select from a "controlled vocabulary," or set of options, but also advantageous to have that vocabulary be able to be updated with new options as needed. Consider:

```
CREATE TABLE sports (
    email VARCHAR(100) NOT NULL,
    sport VARCHAR(100) NOT NULL,
    PRIMARY KEY (email, sport)
);

CREATE TABLE options (
    sport VARCHAR(100) PRIMARY KEY,
);
```

Create PostgreSQL triggers and/or constraints that enforce that each `sports.sport` is one choice from `options.sport`, prohibiting changes to both `sports` and `options` that would violate these constraints:

1. Additions to `sports` must match one of the possible sports in the `options` table.
2. Deletions from and updates to `options` should not be able to remove a sport that's used by some row of `sports` already.

Where possible, avoid triggers when SQL constraints will suffice.

For any triggers that you write, your answer should consist of both a `CREATE FUNCTION` and a `CREATE TRIGGER` statement. Your function definition should be complete, i.e., it should include all pieces (language specification, `BEGIN`, `END`, etc.) and not just the body of the function.

1. Add the following constraint to the `CREATE TABLE` statement for `sports`:
   `FOREIGN KEY (sport) REFERENCES options(sport)`

2.
```
CREATE OR REPLACE FUNCTION delete_from_options()
RETURNS TRIGGER AS
$delim$
BEGIN
      IF(SELECT COUNT(*)
         FROM sports
         WHERE sport = OLD.sport) > 0
      THEN
```

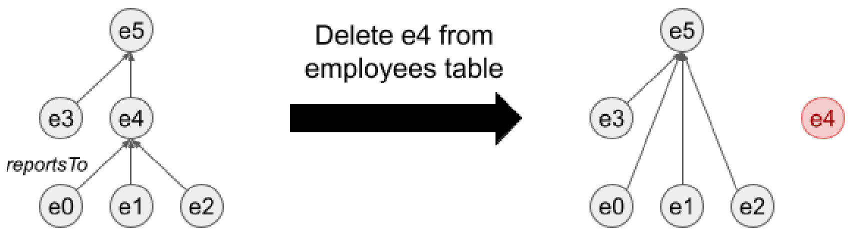
```
            RAISE EXCEPTION ‘cannot delete sport in sports table’;
      END IF;
      RETURN NULL;
END
$delim$
LANGUAGE plpgsql;

CREATE TRIGGER delete_from_options
BEFORE DELETE OR UPDATE ON options
FOR EACH STATEMENT
EXECUTE PROCEDURE delete_from_options()
```

Problem 5 (15 points)

Consider the employees table from the PracticeDB.sql database we used in class. In this table, the `reportsTo` column references another employee, and represents who each employee reports to in the organization (i.e., their supervisor).

Write a SQL trigger so that when an employee is deleted from the table, everyone who reported to that employee has their supervisor changed to the now deleted employee's supervisor:



Your answer should consist of both a `CREATE FUNCTION` and a `CREATE TRIGGER` statement. Your function definition should be complete, i.e., it should include all pieces (language specification, `BEGIN`, `END`, etc.) and not just the body of the function.

Hints:

- You will need to be careful about whether you use a `ROW`-level or `STATEMENT`-level trigger, and whether the trigger is run `BEFORE` or `AFTER` the deletion.
- Modifying an existing row requires using `UPDATE`.
- There are foreign key constraints in place that can prevent employees from being deleted. Make sure that your trigger accounts for them.

```
CREATE OR REPLACE FUNCTION update_supervisor_on_delete()
RETURNS TRIGGER AS
$delim$
BEGIN
      UPDATE employees
      SET reportsTo = OLD.reportsTo
      WHERE reportsTo = OLD.employeeNumber;

      UPDATE customers
      SET salesRepEmployeeNumber = OLD.reportsTo
      WHERE salesRemEmployeeNumber = OLD.employeeNumber;

      RETURN OLD;
```

```
END;
$delim$
LANGUAGE plpgsql;

CREATE TRIGGER update_supervisor_on_delete
BEFORE DELETE ON employees
FOR EACH ROW
EXECUTE PROCEDURE update_supervisor_on_delete()
```

Problem 6 (10 points)

In class, we considered the following table and view:

```
CREATE TABLE audit_items (
  name VARCHAR(100) PRIMARY KEY,
  description TEXT,
  price NUMERIC(10,2),
  -- When the row was created:
  created TIMESTAMP,
  -- When the row was last modified:
  modified TIMESTAMP,
  -- Who last modified the row:
  who VARCHAR(100),
);
```

```
CREATE VIEW items AS
  SELECT name, description, price
  FROM audit_items;
```

And we made this an updatable view for `INSERT` statements using a PostgreSQL rule:

```
CREATE RULE audit_insert AS
ON INSERT TO items DO INSTEAD
INSERT INTO audit_items
VALUES (NEW.name, NEW.description, NEW.price,
        NOW(), NOW(), current_user);
```

Write a PostgreSQL rule (*not* a trigger) to make `items` an updatable view for `UPDATE` statements.

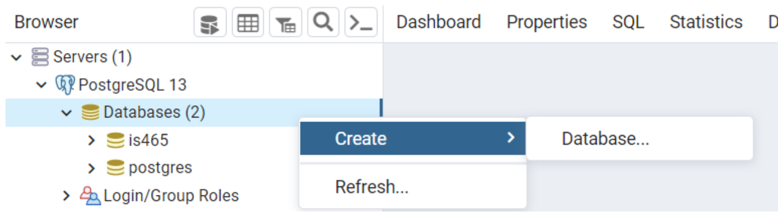
```
CREATE RULE audit_update AS
ON UPDATE TO items DO INSTEAD
UPDATE audit_items
SET name        = NEW.name,
    description = NEW.description,
    price       = NEW.price,
    Modified    = NOW()
    who         = current_user;
WHERE name = OLD.name;
```

Problem 7 (30 points)

Database Setup

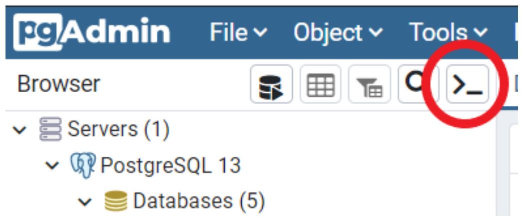
You are given a database as defined in the file `ps3-setup.psql`. Download this file from Canvas. To build this database, open pgAdmin and then:

1. Right-click (or Command+click on macOS) on Databases > Create > Database.



2. In the General tab, enter *ps3* in the Database name field. Click Save.

3. Click on the new ps3 database to connect to it. With the ps3 database selected, click on the PSQL Tool icon.



4. When the psql shell opens, change directory to the location where ps3-setup.psql is located. For me (using Windows 10), it was as follows:

   ```
   \cd 'C:\\Users\\cody\\Downloads'
   ```

   Notice that it is surrounded by quotes and *escapes* the backslashes by using two of them. On macOS, you can use forward slashes in the path instead of backslashes:

   ```
   \cd '//Users//[NAME]//Desktop//'
   ```

5. Import the database:

   ```
   \i ps3-setup.psql
   ```

This database is a recent copy of the USDA nutrition database for foods. It is specifically used to test PostgreSQL performance. Please feel free to explore it.

Instructions

The game in this homework is to make queries run as fast as possible by adding indexes to the tables. The rules of engagement include:

- You may create `BTREE` and `HASH` indexes.
- You may create these on any table, and with any set of columns.
- You may not change the data to make an index work. In other words, you cannot create a `UNIQUE` index on non `UNIQUE` data.
- You cannot change how the query is written, nor its result.

Your goal is to find the least estimated execution time according to `EXPLAIN`. For full credit, your answer to each problem must have the following four pieces of information:

1. `CREATE INDEX` statements showing which indexes you propose to use
2. An explanation for why you chose each of the indexes
3. The output of `EXPLAIN` for the query
4. Whether you ultimately saw a performance increase for the query

For example, for speeding up this query:

```
SELECT name
FROM Movie
WHERE year > 2000;
```

This could be your answer:

Indexes:	`CREATE INDEX year_btree ON Movie(year);`
Reasoning:	The query is looking for all years greater than the year 2000, so a BTREE is a better choice than a HASH
`EXPLAIN`:	`<insert query plan here>`
Speedup?	Yes, the `year_btree` index was used and query performance was improved

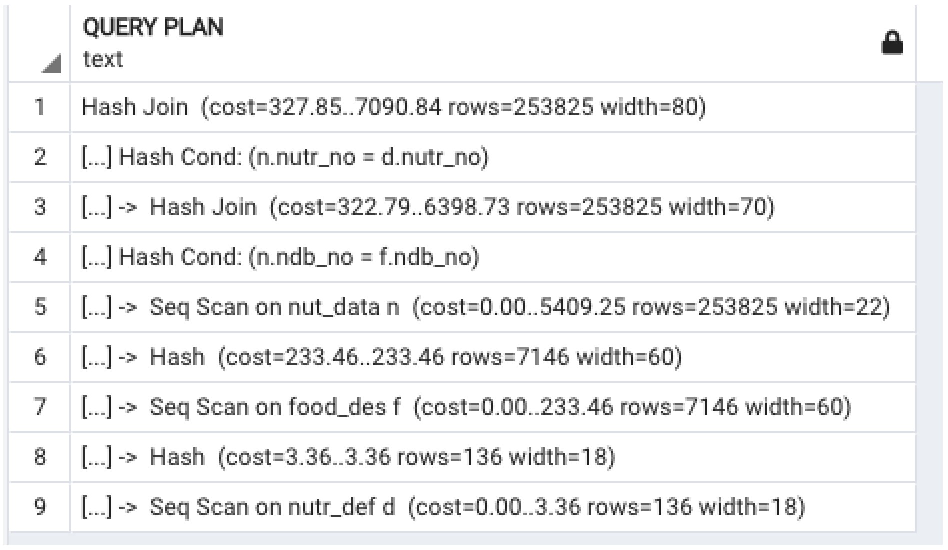
Notes:

- Not all queries will be able to be sped up. Because of the principles we discussed in class -- including the characteristics of the data, distribution of values, size of tables, your computer's capabilities, etc. -- indexing might not change the output of `EXPLAIN`.
- You'll know whether your indexes are making a difference based upon whether `EXPLAIN` actually uses them.
- Even if you were not able to speed up a query, your answer must still list the above four pieces of information.

Problems

Problem a.

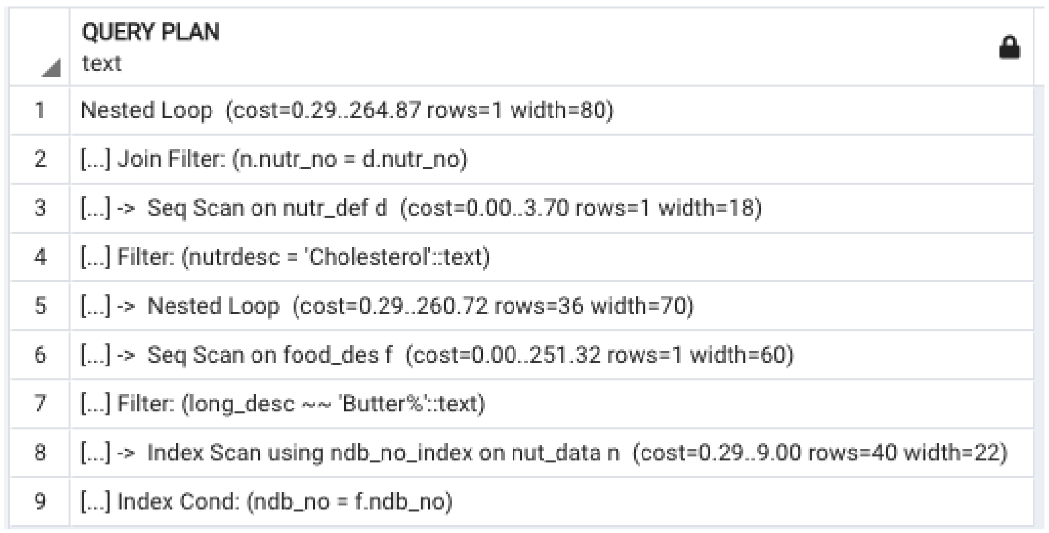
```
SELECT f.long_desc, d.nutrdesc, n.nutr_val, d.units, n.num_studies
FROM food_des f NATURAL JOIN nut_data n NATURAL JOIN nutr_def d;
```

Indexes:	`CREATE INDEX ndb_no_index ON nut_data(ndb_no);` `CREATE INDEX nutr_no_index ON nut_data(nutr_no);`
Reasoning:	`nut_data(ndb_no)` and `nut_data(nutr_no)` are FKs and are both used in the Natural Joins, so a BTREE is a better choice than a HASH.
`EXPLAIN`:	 QUERY PLAN text 1 Hash Join (cost=327.85..7090.84 rows=253825 width=80) 2 [...] Hash Cond: (n.nutr_no = d.nutr_no) 3 [...] -> Hash Join (cost=322.79..6398.73 rows=253825 width=70) 4 [...] Hash Cond: (n.ndb_no = f.ndb_no) 5 [...] -> Seq Scan on nut_data n (cost=0.00..5409.25 rows=253825 width=22) 6 [...] -> Hash (cost=233.46..233.46 rows=7146 width=60) 7 [...] -> Seq Scan on food_des f (cost=0.00..233.46 rows=7146 width=60) 8 [...] -> Hash (cost=3.36..3.36 rows=136 width=18) 9 [...] -> Seq Scan on nutr_def d (cost=0.00..3.36 rows=136 width=18)
Speedup?	No, neither BTREE index was used so query performance did not improve

Problem b.

```
SELECT f.long_desc, d.nutrdesc, n.nutr_val, d.units, n.num_studies
FROM food_des f NATURAL JOIN nut_data n NATURAL JOIN nutr_def d
WHERE f.long_desc LIKE 'Butter%' AND d.nutrdesc = 'Cholesterol';
```

Indexes:	`Same as part (a).`
	 `CREATE INDEX long_desc_index ON food_des(long_desc);`
	 `CREATE INDEX nutrdesc_index ON nutr_def USING HASH (nutrdesc);`

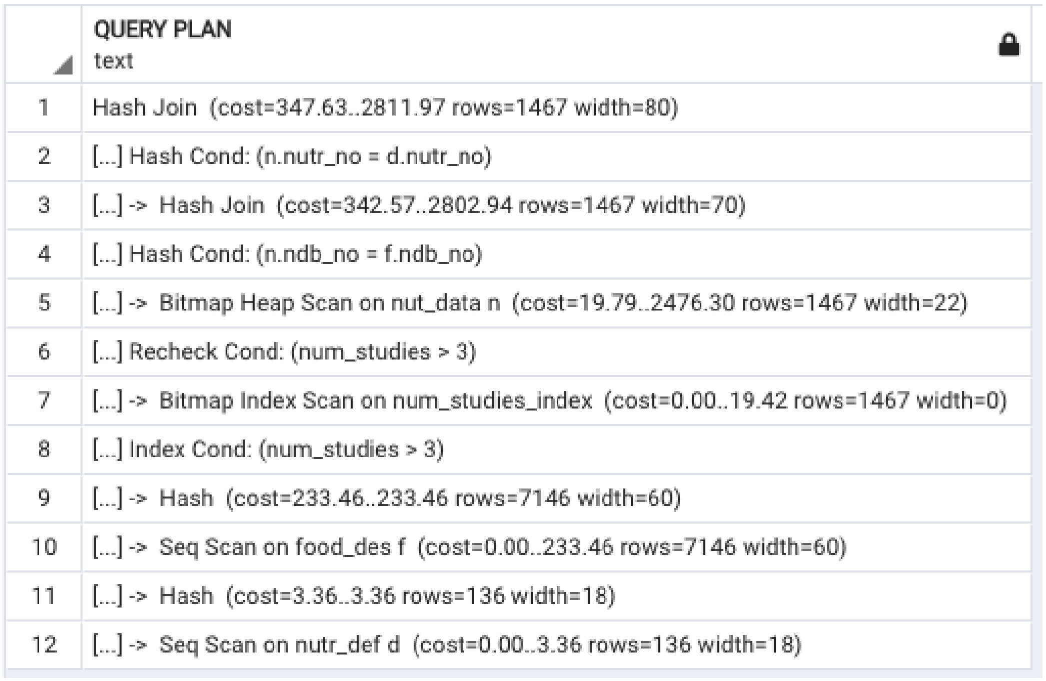
Reasoning:	`food_des(long_desc)` is being queried with a constant pattern at the beginning, so a BTREE would be a better choice than a HASH. `nutr_def(nutrdesc)` is being queried with an equality check, so a HASH is a better choice than a BTREE.
EXPLAIN:	
Speedup?	Yes, the `ndb_no` index was used and query performance improved

```
QUERY PLAN
text
Nested Loop  (cost=0.29..264.87 rows=1 width=80)
[...] Join Filter: (n.nutr_no = d.nutr_no)
[...] ->  Seq Scan on nutr_def d  (cost=0.00..3.70 rows=1 width=18)
[...] Filter: (nutrdesc = 'Cholesterol'::text)
[...] ->  Nested Loop  (cost=0.29..260.72 rows=36 width=70)
[...] ->  Seq Scan on food_des f  (cost=0.00..251.32 rows=1 width=60)
[...] Filter: (long_desc ~~ 'Butter%'::text)
[...] ->  Index Scan using ndb_no_index on nut_data n  (cost=0.29..9.00 rows=40 width=22)
[...] Index Cond: (ndb_no = f.ndb_no)
```

Problem c.

```
SELECT f.long_desc, d.nutrdesc, n.nutr_val, d.units, n.num_studies
FROM food_des f NATURAL JOIN nut_data n NATURAL JOIN nutr_def d
WHERE n.num_studies > 3;
```

Indexes:	`Same as part (a).`
	
	`CREATE INDEX num_studies_index ON nut_data (num_studies)`

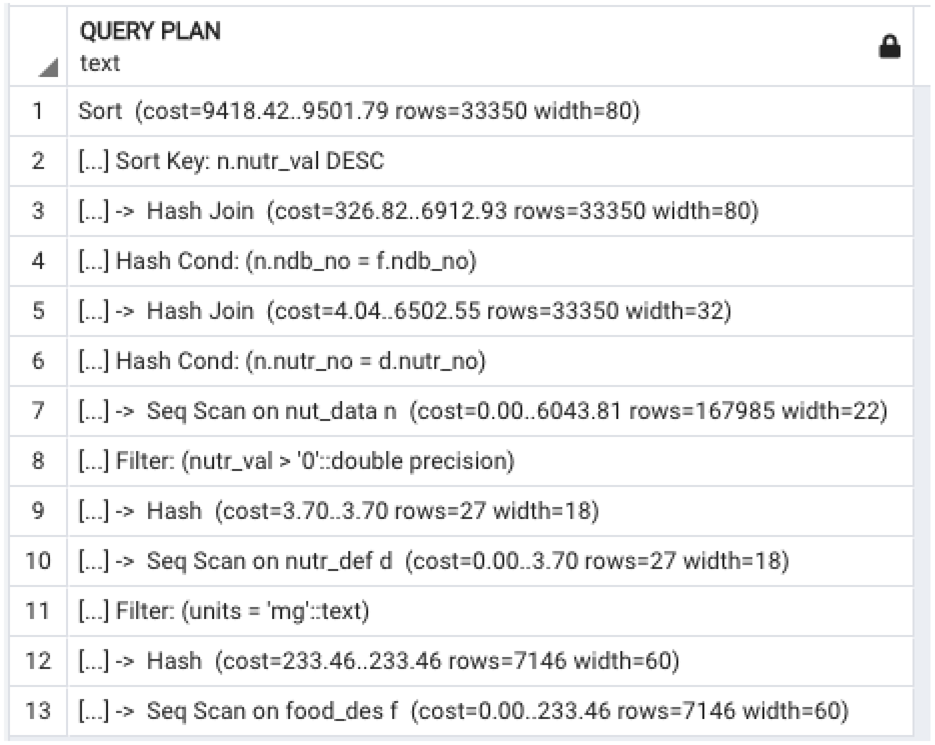
Reasoning:	The query is looking for all rows where `num_studies` is greater than three (i.e. a range), so a BTREE is a better choice than a HASH.
EXPLAIN:	
Speedup?	Yes, the `num_studies` index was used and query performance improved

```
QUERY PLAN
text
1   Hash Join  (cost=347.63..2811.97 rows=1467 width=80)
2   [...] Hash Cond: (n.nutr_no = d.nutr_no)
3   [...] ->  Hash Join  (cost=342.57..2802.94 rows=1467 width=70)
4   [...] Hash Cond: (n.ndb_no = f.ndb_no)
5   [...] ->  Bitmap Heap Scan on nut_data n  (cost=19.79..2476.30 rows=1467 width=22)
6   [...] Recheck Cond: (num_studies > 3)
7   [...] ->  Bitmap Index Scan on num_studies_index  (cost=0.00..19.42 rows=1467 width=0)
8   [...] Index Cond: (num_studies > 3)
9   [...] ->  Hash  (cost=233.46..233.46 rows=7146 width=60)
10  [...] ->  Seq Scan on food_des f  (cost=0.00..233.46 rows=7146 width=60)
11  [...] ->  Hash  (cost=3.36..3.36 rows=136 width=18)
12  [...] ->  Seq Scan on nutr_def d  (cost=0.00..3.36 rows=136 width=18)
```

Problem d.

```
SELECT f.long_desc, d.nutrdesc, n.nutr_val, d.units, n.num_studies
FROM food_des f NATURAL JOIN nut_data n NATURAL JOIN nutr_def d
WHERE d.units='mg' AND n.nutr_val > 0
ORDER BY n.nutr_val DESC;
```

Indexes:	`Same as part (a).`
	
	`CREATE INDEX units_index ON nutr_def USING HASH (units)`
	`CREATE INDEX nutr_val_index ON nut_data (nutr_val)`

Reasoning:	The query is looking for all rows with a `unit` of mg (i.e. equality), so a HASH is a better choice than a BTREE. The query is looking for all rows with `nutr_val` greater than 0 (i.e. a range), so a BTREE is a better choice than a HASH. The query is ordering all the results by `nutr_val`, so again a BTREE is a better choice than a HASH.
EXPLAIN:	 QUERY PLAN text 1 Sort (cost=9418.42..9501.79 rows=33350 width=80) 2 [...] Sort Key: n.nutr_val DESC 3 [...] -> Hash Join (cost=326.82..6912.93 rows=33350 width=80) 4 [...] Hash Cond: (n.ndb_no = f.ndb_no) 5 [...] -> Hash Join (cost=4.04..6502.55 rows=33350 width=32) 6 [...] Hash Cond: (n.nutr_no = d.nutr_no) 7 [...] -> Seq Scan on nut_data n (cost=0.00..6043.81 rows=167985 width=22) 8 [...] Filter: (nutr_val > '0'::double precision) 9 [...] -> Hash (cost=3.70..3.70 rows=27 width=18) 10 [...] -> Seq Scan on nutr_def d (cost=0.00..3.70 rows=27 width=18) 11 [...] Filter: (units = 'mg'::text) 12 [...] -> Hash (cost=233.46..233.46 rows=7146 width=60) 13 [...] -> Seq Scan on food_des f (cost=0.00..233.46 rows=7146 width=60)
Speedup?	No, neither index was used so query performance did not improve.

Problem e.

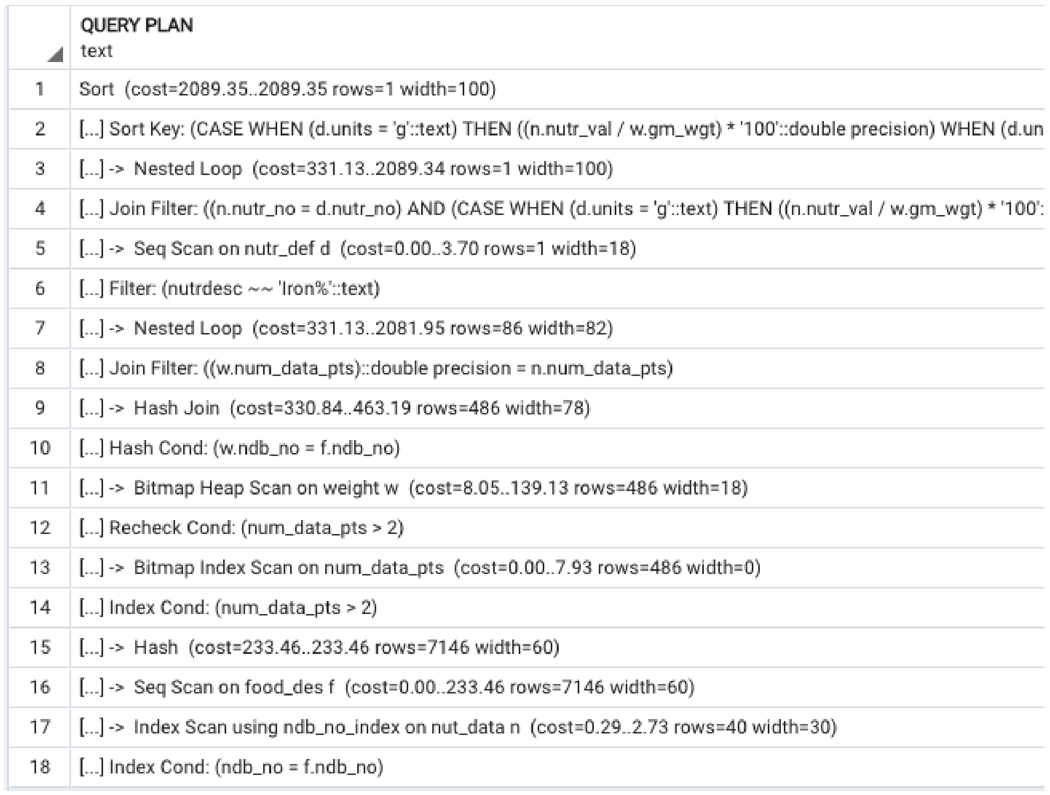
The following query uses a `CASE` expression in the `SELECT` clause, which is essentially an if/else if/else conditional statement.

```
CREATE VIEW percentages AS
SELECT f.long_desc, d.nutrdesc, n.nutr_val, d.units,
       n.num_studies, w.gm_wgt, w.num_data_pts,
  CASE WHEN d.units = 'g' THEN (n.nutr_val / w.gm_wgt) * 100
       WHEN d.units = 'mg' THEN ((n.nutr_val / 1000) / w.gm_wgt) * 100
       WHEN d.units LIKE 'mcg%' THEN ((n.nutr_val / 1000000)/ w.gm_wgt) * 100
       ELSE NULL
  END AS percent
FROM food_des f NATURAL JOIN nut_data n NATURAL JOIN nutr_def d
     NATURAL JOIN weight w;

SELECT *
FROM percentages
WHERE nutrdesc LIKE 'Iron%'
  AND percent < 1
  AND num_data_pts > 2
ORDER BY percent;
```

Indexes:	`Same as part (a).`
	`CREATE INDEX ndb_no_index ON weight (ndb_no)`
	
	`CREATE INDEX num_data_pts ON weight (num_data_pts)`

Reasoning:	`weight(ndb_no)` is a FK, so a BTREE is a better choice than a HASH. The query is looking for all rows where `num_data_pts` is less than 2 (i.e. a range), so a BTREE is a better choice than a HASH.

EXPLAIN:	

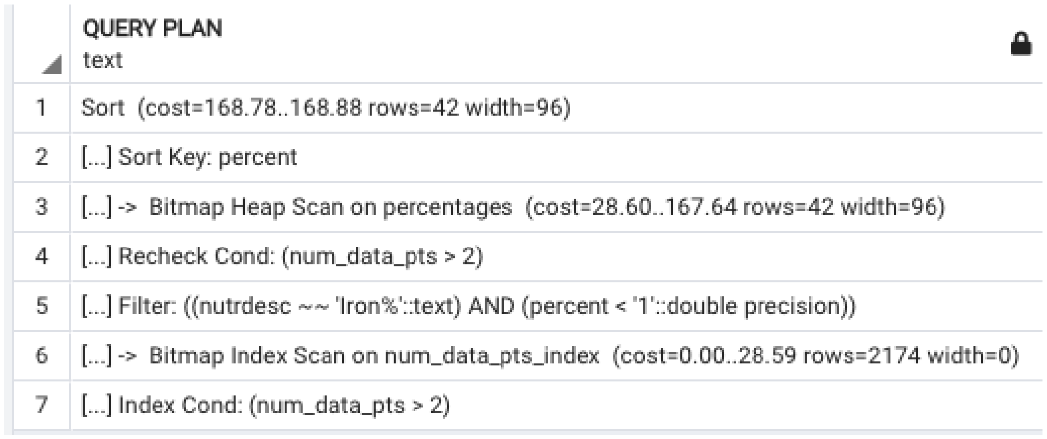
```
    QUERY PLAN
    text
1   Sort  (cost=2089.35..2089.35 rows=1 width=100)
2   [...] Sort Key: (CASE WHEN (d.units = 'g'::text) THEN ((n.nutr_val / w.gm_wgt) * '100'::double precision) WHEN (d.un
3   [...] ->  Nested Loop  (cost=331.13..2089.34 rows=1 width=100)
4   [...] Join Filter: ((n.nutr_no = d.nutr_no) AND (CASE WHEN (d.units = 'g'::text) THEN ((n.nutr_val / w.gm_wgt) * '100':
5   [...] ->  Seq Scan on nutr_def d  (cost=0.00..3.70 rows=1 width=18)
6   [...] Filter: (nutrdesc ~~ 'Iron%'::text)
7   [...] ->  Nested Loop  (cost=331.13..2081.95 rows=86 width=82)
8   [...] Join Filter: ((w.num_data_pts)::double precision = n.num_data_pts)
9   [...] ->  Hash Join  (cost=330.84..463.19 rows=486 width=78)
10  [...] Hash Cond: (w.ndb_no = f.ndb_no)
11  [...] ->  Bitmap Heap Scan on weight w  (cost=8.05..139.13 rows=486 width=18)
12  [...] Recheck Cond: (num_data_pts > 2)
13  [...] ->  Bitmap Index Scan on num_data_pts  (cost=0.00..7.93 rows=486 width=0)
14  [...] Index Cond: (num_data_pts > 2)
15  [...] ->  Hash  (cost=233.46..233.46 rows=7146 width=60)
16  [...] ->  Seq Scan on food_des f  (cost=0.00..233.46 rows=7146 width=60)
17  [...] ->  Index Scan using ndb_no_index on nut_data n  (cost=0.29..2.73 rows=40 width=30)
18  [...] Index Cond: (ndb_no = f.ndb_no)
```

Speedup?	Yes, the `num_data_pts` index and the `nbd_no` index were both used and query performance improved.

Problem f.

Repeat Problem e, but this time *materialize* the view. This is a matter of writing `CREATE MATERIALIZED VIEW` Instead of `CREATE VIEW` In the above SQL. You will need to `DROP VIEW percentages;` first.

Indexes:	`CREATE INDEX percent_index ON percentages (percent)`
	`CREATE INDEX num_data_pts_index ON percentages (num_data_pts)`
	`CREATE INDEX nutrdesc_index ON percentages (nutrdesc)`

Reasoning:	The query is looking for all rows where percent is less than 1 (i.e. a range) and will order the results by percent, so a BTREE is a better choice than a HASH. The query is looking for all rows where `num_data_pts` is less than 2 (i.e. a range), so a BTREE is a better choice than a HASH. The query is looking for all rows where `nutrdesc` has a beginning pattern, so a BTREE is a better choice than a HASH
`EXPLAIN`:	 QUERY PLAN text 1 Sort (cost=168.78..168.88 rows=42 width=96) 2 [...] Sort Key: percent 3 [...] -> Bitmap Heap Scan on percentages (cost=28.60..167.64 rows=42 width=96) 4 [...] Recheck Cond: (num_data_pts > 2) 5 [...] Filter: ((nutrdesc ~~ 'Iron%'::text) AND (percent < '1'::double precision)) 6 [...] -> Bitmap Index Scan on num_data_pts_index (cost=0.00..28.59 rows=2174 width=0) 7 [...] Index Cond: (num_data_pts > 2)
Speedup?	Yes, the `num_data_pts` index was used and query performance improved.