The purpose of this problem set is to give you some practice with concepts related to schema design, query planning, and query processing. Start early as this assignment is long.

Part 1 - Query Plans and Cost Models

In this part of the problem set, you will examine query plans that PostgreSQL uses to execute queries, and try to understand why it produces the plan it does for a certain query.

We are using the same database as for Problem Set 1. The command to connect has changed slightly, however:

```sql
psql -h geops.csail.mit.edu -p 5433 wikidb
```

Make sure your PostgreSQL client is 9.3+ so that your results are consistent with the solutions. Athena already has client version 9.3.9 installed, so you can simply ssh into athena.dialup.mit.edu and get started. In case you want to work on your own Debian/Ubuntu machine, you can install the `postgresql-client` package by running the following command in your shell.

```sh
sudo apt-get install postgresql-client
```

To help understand database query plans, PostgreSQL includes the `EXPLAIN` command. It prints the physical query plan for the input query, including all of the physical operators and internal access methods being used. For example, the SQL command displays the query plan for a very simple query:

```sql
postgres@wikidb# explain select * from page;
```

```sql
QUERY PLAN
---------------------------------------------------------------
Seq Scan on page (cost=0.00..10329.80 rows=374880 width=106)
(1 row)
```

To be able to interpret plans like the one above, you should refer to the `explain basics` section in the Postgres documentation.

We have run `VACUUM FULL ANALYZE` on all of the tables in the database, which means that all of the statistics used by PostgreSQL server should be up to date.
1. **[1 points]**: Which indices exist for table `page` in `wikidb`? **Answer:**
There is one on `page_id` and one on `(page_namespace, page_title)`.

2. **[2 points]**: Which query plan does postgres choose for `select page_id from page`? Is it different from the plan shown in the previous page? Given the indices we have defined on our table, are there any other possible query plans? **Answer:**
Postgres picks a sequential scan on the page.

```
Seq Scan on page
(cost=0.00..10329.80 rows=374880 width=8)
```

3. **[1 points]**: In one sentence, describe the difference between the plan from the previous question and the plan for query: `select page.id from page order by page.id`. **Answer:**
Postgres scans the index rather than the file in this case.

```
Index Only Scan using page_page_id_idx on page
(cost=0.42..35757.50 rows=374880 width=8)
```

4. **[3 points]**: Both of the queries above (`select page.id from page` and `select page.id from page order by page.id`) need to only use the value of a column which has already been indexed. Because of this, Ben Bitdiddle is convinced that both these queries run fastest using an index-only scan. Can you use Postgres to provide some evidence that confirms or disproves Ben’s theories about access methods? **Answer:**
The answer you get here turns out to be trickier than we wanted it. What you answered depends partly on the right on the internal state of the database when you ran the query, so we accept different answers as long as you put in the numeric evidence of what ran faster, but we don’t expect you to know that.

We can coerce postgres into using a variety of query plans by using the `enable_seqscan`, `enable_indexonlyscan`, `enable_indexscan` and the `enable_bitmapscan` flags.

```
explain analyze select page_id from page
-- seq scan
Seq Scan on page
(cost=0.00..10329.80 rows=374880 width=8)
(actual time=0.006..71.978 rows=374880 loops=1)
Planning time: 0.034 ms
Execution time: 90.141 ms

-- index scan without vacuum.

Index Only Scan using page_page_id_idx on page
(cost=0.42..35757.50 rows=374880 width=8)
(actual time=0.018..250.813 rows=374880 loops=1)
Heap Fetches: 374880
Planning time: 0.071 ms
```
Execution time: 269.400 ms

-- index scan with vacuum: pure index scan.

Index Only Scan using page_page_id_idx on page
(cost=0.42..9747.62 rows=374880 width=8)
(actual time=0.017..57.930 rows=374880 loops=1)
    Heap Fetches: 0
    Planning time: 0.049 ms
    Execution time: 76.323 ms

So, depending on the state of the database your answer may be 'index scans are actually slower' or 'index scans are faster', and we will consider it correct as long as you show timing evidence.

For the ordered select query, forcing the database to use a SeqScan also forces it to do a final sort. So the index only scan with our without vacuum is a clear win.

```
explain analyze select page_id from page order by page_id
```

Index Scan using page_page_id_idx on page
(cost=0.42..35757.50 rows=374880 width=8)
(actual time=0.022..279.928 rows=374880 loops=1)
    Planning time: 0.080 ms
    Execution time: 298.505 ms
(5 rows)

5. [3 points]: Consider the two following queries and their plans from wikidb:

```
wikidb=> explain analyze select page_namespace, page_title from page where page_title = 'abc';
```

```
QUERY PLAN
------------------------------------------------------------------
| Index Only Scan using page_page_namespace_page_title_idx on page |
| (cost=0.42..10332.03 rows=1 width=20) (actual time=31.226..31.226 rows=0 loops=1) |
| Index Cond: (page_title = 'abc'::text) |
| Heap Fetches: 0 |
| Planning time: 0.107 ms |
| Execution time: 31.264 ms |
(5 rows)
```

```
wikidb=> explain analyze select page_namespace, page_title from page where page_namespace = 11;
```

```
QUERY PLAN
------------------------------------------------------------------
| Index Only Scan using page_page_namespace_page_title_idx on page |
| (cost=0.42..308.87 rows=172 width=20) (actual time=0.122..0.399 rows=628 loops=1) |
| Index Cond: (page_namespace = 11) |
| Heap Fetches: 628 |
| Planning time: 0.093 ms |
| Execution time: 0.460 ms |
(5 rows)
```

The two queries and their plans are very similar and make use of the same index. Why are the costs (both the estimates and actual) so different?
The two queries are similar, but the order of the index key matters. The index on \((\text{page_namespace}, \text{page_title})\) will have great locality with respect to pages in the same \text{page_namespace}; all pages with the same namespace will be physically close together on the index, on the other hand pages with the same \text{page_title} will be scattered through the index, therefore we have to scan much more of the index in order get them.

Now consider the queries generated by replacing \$0\ with 10, 20 and 30 in the following template. The query computes the number of links between pages of different sizes. (You can call the three queries Q10, Q20 and Q30 respectively).

```sql
explain select p0.page_id, p1.page_id from page as p0, pagelinks as pl01, page as p1
where (p0.page_id = pl01.pl_from)
and (pl01.pl_title, pl01.pl_namespace) = (p1.page_title, p1.page_namespace)
and p0.page_len > 4000 and p1.page_len < \$0;  
```

6. [2 points]: What physical plan does PostgreSQL use for each of them? Your answer should consist of a drawing of the three query trees and annotations on each node. \textit{Answer:}

see Figure

7. [1 points]: Which access methods are used? (also label them in the diagrams) \textit{Answer:}

see Figure

8. [1 points]: Which join algorithms? (also label them in the diagrams) \textit{Answer:}

see Figure

9. [2 points]: By running some queries to compute the sizes of the intermediate results in the query, and/or using \texttt{EXPLAIN ANALYZE}, can you see are there any final or intermediate results where PostgreSQL’s estimate is less than half (or more than double) the actual size? \textit{Answer:}

see Figure

10. [4 points]: At which values of \texttt{p1.page_len} (in the range of 0 to 30) do the plans change? Do you believe the query planner is switching at the correct points? (justify your answer quantitatively).

\textit{Answer:}

The first switch point happens when \$0 = 15\. The actual runtime switches from 165ms to 265ms. This is a strong hint that postgres switched early, but we don’t know if there simply are lots of results at \$0 = 14\ that weren’t there before, for example. We can clarify which plan is actually better if we coerce postgres to use the same plan as before instead of switching, the runtime is in the range of 190ms, still below the 265ms.

The other switch point happens at \$0 = 28\, at which point it uses a sort-merge join that takes 27000 seconds. Again, the runtime jumps, so Postgres is switching too early. If we coerce postgres to not use the merge join, it picks a hash join that runs in 1 second.
Part 2 – Query Plans and Access Methods

In this problem, your goal is to estimate the cost of different query plans and think about the best physical query plan for a SQL expression.

You have been hired to optimize the database of a hot new startup, “yomomma.com”, where users can send their friends “Yo’ Momma” jokes (these are jokes like “Your momma’s so stinky, I thought she was cheese.”).

The interface to the application is very simply – you can invite and add friends, and then tap on a friend to send them a new joke they have not received before. The website maintains a large and growing database of jokes. Friends in yomomma are symmetric (i.e., if A is B’s friend, B is A’s friend), and this friendship will be represented by two tuples in the friends table. When user receives a joke, both the user and their friends can choose to “like” it.

The database contains the following tables:

```sql
-- user u_uid with a name, gender, and birthday
CREATE TABLE users(
   u_id int PRIMARY KEY,
   u_name char(50),
   u_gender char,
   u_bday date);

-- table indicating f_uid1 and f_uid2 are friends, and the date they became friends
-- if table contains a friendship pair (X,Y), it will also include the pair (Y,X)
CREATE TABLE friends(
   f_ship_id int PRIMARY KEY,
   f_uid1 int REFERENCES users(u_id),
   f_uid2 int REFERENCES users(u_id),
   f_friended date);
```
-- table of jokes
CREATE TABLE jokes(
    j_id int PRIMARY KEY,
    j_joke char(400) --all jokes are allocated to be 400 chars
);

-- table of jokes sent, from one user to another
CREATE TABLE sent(
    s_id int PRIMARY KEY,
    s_from int REFERENCES users(u_id),
    s_to int REFERENCES users(u_id),
    s_j_id int REFERENCES jokes(j_id)
);

-- table of likes, by a particular user, of a particular sent joke
CREATE TABLE likes(
    l_id int PRIMARY KEY,
    l_s_id int REFERENCES sent(s_id),
    l_uid int REFERENCES users(u_id)
);
In this database, int and date values are 8 bytes each and characters are 1 byte. All tuples have an additional 8 byte header. This means, that, for example, the size of a single joke record is $8 + 8 + 400 = 416$ bytes.

You create these tables in a row-oriented database. The system supports heap files and B+-trees (clustered and unclustered). B+-tree leaf pages point to records in the heap file. Assume you can cluster each heap file in according to exactly one B+tree, and that the database system has up-to-date statistics on the cardinality of the tables, and can accurately estimate the selectivity of every predicate. Assume B+-tree pages are 50% full, on average. Assume disk seeks take 10 ms, and the disk can sequentially read 100 MB/sec. In your calculations, you can assume that I/O time dominates CPU time (i.e., you do not need to account for CPU time.)

For the queries below, you are given the following statistics:

<table>
<thead>
<tr>
<th>Statistic</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of users</td>
<td>$10^6$</td>
</tr>
<tr>
<td>Number of friends</td>
<td>$10^7$</td>
</tr>
<tr>
<td>Number of jokes</td>
<td>$10^4$</td>
</tr>
<tr>
<td>Number of jokes sent</td>
<td>$10^8$</td>
</tr>
<tr>
<td>Number of likes</td>
<td>$10^9$</td>
</tr>
</tbody>
</table>

In the absence of other information, assume that attribute values are uniformly distributed (e.g., that there are approximately the same number of friends per user, jokes sent per user, likes per joke, etc).

Suppose you are running the query `SELECT COUNT(*) FROM likes WHERE l_id = 1`. Answer the following:

11. [1 points]: In one sentence, what would be the best plan for the DBMS to use assuming no indexes? Approximately how long would this plan take to run, using the costs and table sizes given above?

Answer:
The only option is to use a sequential scan, but we still have the possibility that the data is sorted according to l_id or not. If the data is sorted according to l_id and all ids are positive, then it would be only one read to the first page. Reading a single page consists of seeking (10ms) and then scanning that page. Assuming a page size of 4 KB, the scan time at 100MB/s is negligible. 

If the data is not sorted according to l_id, then we must scan on-average 1/2 of the table. A row is of size (4 * 8) bytes, and the table has $10^9$ rows, for 32GB of data. At 100MB/s, scanning takes 160 seconds. This cost dominates over an initial seek.

12. [2 points]: In one sentence, what would be the best plan for the DBMS to use assuming a clustered B+tree index on l_id? Approximately how long would this plan take?

Answer:
Assuming 4KB pages, and 8 byte keys + 8 byte pointers for each entry then each page holds $4KB/16B = 2^8$ entries. At half capacity, this means $2^7$ entries. So, the branching factor (accounting for half full pages) is $2^7$. The depth of the tree then is $\log_{2^8} 30 = 30/7 = 5$. Holding 4 levels of the tree in memory consumes $4KB * (128)^4 = 8GB$, so we see it as a reasonable assumption. But holding one more level would consume > 1TB, so we only assume we hold up to 4 levels.

Therefore, assuming caching on the first four levels, the IO cost of a BTree lookup is 0 or 1, followed by accessing the actual Heap file. So, total is 1 or 2 seeks, 10 or 20ms.

13. [2 points]: In one sentence, what would be the best plan for the DBMS to use assuming an unclustered B+tree index on l_id? Approximately how long would this plan take?
Figure 2: A possible query plan with estimated cardinalities. We have omitted projection operators. See answers for details on how cardinalities were derived.

**Answer:** 
The cost of traversing the clustered tree is the same in this case because we are already doing the minimal amount of traversal work. If we wanted to do a query for a range of ids, then the answer changes.

Now consider the following query. For a given user U, this query computes the number of non-friends of U who 'liked' a joke sent by U.

```
SELECT COUNT(*)
FROM likes AS l, sent AS s, users AS u
WHERE l.l_uid NOT IN (SELECT f_uid2
                      FROM friends AS f
                      WHERE f.f_uid1 = u.u_id)
       AND u.u_name = 'Bob Smith'
       AND u.u_id = s.s_from
       AND l.l_s_id = s.s_id
```

Note that the subquery references the `u.u_id` value from the outer query (this value represents the user who sent the joke). One possible way to execute this subquery is to simply run a separate subquery over the `friends` table for each like. You can represent this in a query plan by a select node where the select condition is a subquery (you may be able to come up with a more efficient plan but this is not required.)

14. **3 points**: Suppose only heap files are available (i.e., there are no indexes), and that the system supports grace (hybrid) hash, merge join, and nested loops joins. For each node in your query plan indicate (on the drawing, if you wish), the approximate output cardinality (number of tuples produced.)

**Answer:** 
For the selectivities in the diagram. Assuming 1 users with name Bob Smith:
Joining selected users with sent:

10^8 (sent) / 10^6 (user) → 100 jokes / user * 1 user ≈ 100 jokes total.

Joining likes with (jokes, user):

There are 10^9 / 10^8 = 10 likes per joke * 100 jokes = 1000 total likes for those jokes.

Nested subquery of friends for given user id: Most people have 10^7 / 10^6 = 10 friends.

The selection based on NOT IN: for every 10 friends, there are 10^6 - 10 non friends. So, if we really stick to assuming full randomness, 99.999% of users likes are from non-friends.

So, selection is almost guaranteed to succeed, and output cardinality is 1000 still.

Finally, the aggregation returns only one tuple.

15. [2 points]: Estimate the runtime of the plan you drew, in seconds.

Answer:
Scan of users: (8B header + 8B pkey + 50 + 1 + 8) = 75B/row * 10^6 rows = 75 MB. ~ 1 s.

Scan of sent: 40 bytes/row * 10^8 row = 4GB. Time is 4GB / 100 MB /s = 40 s.

The (Sent, User) join can be done as a nested loops join with the selected users as inner loop. The Sent users table simply streams through it only once, so there is no extra IO cost.

For the join of (Sent, User) with Likes: memory hash or nested loops (remember we don’t count cpu here) with the outer loop being likes.

Scan of likes: 32 bytes/row * 10^9 = 32GB. Time is 32GB / 100MB/s = 320 s

So, up to this point of getting all the likes the query has used 360s.

Select subquery: for each (like) output from the previous plan (there is 1000)
Scan of friends: 40 bytes/row * 10^7 rows = 400 MB -> 4s.

To answer the NOT IN predicate, we can do a loop through the selected friends (10). No extra IO cost here. Because we do this 1000 times, runtime is 4s*1000 = 4000s.

In total then the subqueries dominate the rest of the tree and the runtime is ~ 4000s

16. [2 points]: Now, suppose that there are clustered B+Trees on u_id, l_uid, s_from, and f_uid1, and an unclustered B+Tree on u_name, s_id, and l_s_id. Draw the new plan you think the database would use and estimate its runtime methods available to it.

Answer:
Notice that using an index to scan a 4KB page takes 10ms (random IO) + scan time. The scan time is 4KB/100MB/s = 1/25 of a millisecond, which is negligible compared to the random IO.

We’re sticking with the same plan, but now we avoid multiple scans: Lookup table users by u_name using unclustered B+Tree on u_name. There are only a few, so 10 ms.* 1 user = 10ms. (2 users with name Bob Smith would give 20 ms, though).

Lookup table sent with by s_from. This index is clustered, so 10 ms (1 or 2 pages of clustered results add negligible overhead). Lookup table likes using unclustered index on l_s_id, implies 1000 single lookup random IO and 10ms each ~ 10s.

Up to this point in the plan, we have used 10s, dominated by lookups into the likes table.

Select subquery: Clustered index lookup on friends in order to find his friends: 10ms of index followed by short scan. (10 friends).
Because we do this 1000 times, the lookup total time is 10s.

The total time estimate for the query is 20s.

17. [2 points]: Suppose you could choose your own indexes (assuming at most one clustered index per table) for this plan; what would you choose, and why? Justify your answer quantitatively.

Answer:
We see that the total time is dominated by the lookups onto table Likes, and the nested subquery lookups. The high cost of the lookups into likes is because the index is unclustered. If we clustered the likes table by l_s_id then we would lose the chance to cluster by l_uid, but we don’t use that index in this query anyway. This change would decrease the time of lookups into likes to 1 lookup / unique sent joke * 100 sent jokes * 10 ms / lookup = 1s. So the overall query goes down to 10s.

Going beyond indices (not expected in your solution, but in case you were wondering)
If we wanted to improve performance further, we would need to deal with the correlated subquery. One option is to sort the likes we got by user id, and then only do one subquery by batch of groups of likes with the same user id (materilizing and re-using the sub-query result). This would allow us to do the a correlated subquery (10ms) only once per user id. Since there is only about 1 user id, then we would only spend 10ms in the nested subquery. Our runtime would go down to close to 1 second (dominated by lookups into Likes once again). It is unclear how much of this reasoning can be automated in a database.

Alternatively, a feasible query plan that avoids the repeated nested subquery is one that first flattens the nested query using a left outer join, like discussed in class.
Part 3 – Schema Design and Query Execution

A medical lab testing company has several testing centers all over the country. In this problem, you will design a schema to keep track of the testing centers, tests and order information.

Specifically, you will need to keep track of:

1. The address, city, state and manager for each testing center.

2. The equipment at each testing center including the name of the machine, manufacturing year, and status of the machine (Functional, in repair, phased out). Note that one center may have multiple machines of the same kind.

3. The tests that a center runs including the test id, name, type (blood, urine, stool etc), time to run and price. The price of the test may depend on the testing center.

4. The orders for tests including the test id, doctor who submitted the test, the date the test was submitted, the patient for whom the test was ordered.

5. The results of tests including the date the test was run, time taken to run and results. For this problem, assume that each test produces a single real number.

18. [2 points]: Write out a list of functional dependencies for this schema. Not every fact listed above may be expressible as a functional dependency.

Answer:
The functional dependencies are (note: some id fields have been added):

- Testing center id ⇒ Address, City, State, Manager for testing center
- Equipment id ⇒ Name, Manufacturing Year, Status
- Test id ⇒ Name, Type, Time to Run
- Order id ⇒ Doctor, Date submitted, Patient, Test center
- Test id, Testing center id ⇒ Price

19. [3 points]: Draw an ER diagram representing your database. Include a few sentences of justification for why you drew it the way you did.

20. [2 points]: Write out a schema for your database in BCNF. You may include views. Include a few sentences of justification for why you chose the tables you did.

Answer:

Primary keys are underlined:

- TestCenter(tcid, address, city, state, manager)
- Equipment(eid, name, year, status)
- Test(tid, name, type, time_to_run)
• HasEquipment(eid references Equipment.eid, tcid references TestCenter.tcid)

• Runs(tid references Test.tid, tcid references TestCenter.tcid, price)

• Order(oid, date, patient, doctor)

• RunAt(oid references Order.oid, tcid references TestingCenter.tcid)

• Contains(oid reference Order.oid, tid references Test.tid, date_run, time_to_run, result)

21. **[2 points]**: Is your schema redundancy and anomaly free? Justify your answer.

*Answer:*
Yes, since the schema is in BCNF, it is redundancy and anomaly free with respect to functional dependencies.

22. **[3 points]**: Suppose you wanted to ensure that no test can be run without a doctor ordering it. How can you enforce this constraint? In addition, you want to ensure that no more than 10% of active machines (not phased out) in a testing center can be in repair at any time. How can you enforce this constraint?

*Answer:*
The first constraint can be enforced with a primary key - foreign key relationship and not null. The second can be enforced through the use of triggers.