There are 13 questions and 13 pages in this quiz booklet. To receive credit for a question, answer it according to the instructions given. You can receive partial credit on questions. You have 80 minutes to answer the questions.

Write your name on this cover sheet AND at the bottom of each page of this booklet.

Some questions may be harder than others. Attack them in the order that allows you to make the most progress. If you find a question ambiguous, be sure to write down any assumptions you make. Be neat. If we can’t understand your answer, we can’t give you credit!

THIS IS AN OPEN BOOK, OPEN NOTES QUIZ.
LAPTOPS MAY BE USED; NO PHONES OR INTERNET ALLOWED.

Do not write in the boxes below

<table>
<thead>
<tr>
<th>1-4 (xx/22)</th>
<th>5-7 (xx/28)</th>
<th>8-10 (xx/22)</th>
<th>11-13 (xx/28)</th>
<th>Total (xx/100)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Name:
I Column Stores

Suppose you are running this query over a database of employees and departments:

SELECT e.rank, d.bldg, AVG(e.sal)
FROM employees e, departments d
WHERE e.dno = d.dno
GROUP BY e.rank, d.bldg

Suppose there are 10,000 employees, 100 departments, each in a distinct building, and 20 ranks, and that salaries are distributed uniformly between $50,000 and $150,000. Assume ranks are uniformly distributed across departments.

Suppose you are evaluating this query in C-Store.

1. [6 points]: What is the best sort order for the emp and dept tables, given you are running only the above query? Briefly state why.
   
   emp: e.dno, e.rank
   dept: d.dno
   
   Why?
   Sorting both tables on dno allows the join to be completed as a merge, with just a linear scan of both tables. Sorting emp on rank allows the rank column to be compressed significantly, since there are 100 departments x 20 = 2,000 department/rank pairs, and 10,000 employees, so there will still be runs of length 5. The aggregate will have to be performed using hashing, but the hash table should easily fit into memory, as there are just 2,000 groups.
   
   Choosing to sort the tables in rank/building order will require the use of hashing or NL for the join, both of which will incur extra CPU relative to just performing comparisons for the merge. Furthermore, it is not clear what the advantage of this is as there’s no obvious way to get the output of the join sorted in e.rank, d.bldg order without an explicit sort of the join output (if this were possible the aggregate could be performed without hashing). It may seem that sorting emp on rank,dno and dept on bldg (or dno) and then doing an NL join will achieve this but its important to realize that dept sorted on bldg is NOT sorted on dno (despite there being one building per department.) Consider the following example:
   
<table>
<thead>
<tr>
<th>rank</th>
<th>dno</th>
<th>bldg</th>
<th>dno</th>
</tr>
</thead>
<tbody>
<tr>
<td>prof 1</td>
<td>NL</td>
<td>32</td>
<td>3</td>
</tr>
<tr>
<td>prof 2</td>
<td>join</td>
<td>34</td>
<td>1</td>
</tr>
<tr>
<td>prof 3</td>
<td>on</td>
<td>36</td>
<td>2</td>
</tr>
<tr>
<td>stud 2</td>
<td>dno</td>
<td></td>
<td></td>
</tr>
<tr>
<td>stud 3</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

   Sorting dept on dno in the above example doesn’t make the output any better.

   Hence, hashing is going to be required for the aggregate in any case.

Name:
2. [5 points]: Given the sort orders and table statistics given above, indicate which compression method should be used for each column to make the above query as fast as possible, choosing from:

A. gzip (e.g., Lempel-Ziv compression)
B. run-length encoding (RLE)
C. delta encoding

(For each column, indicate your answer (A-E))

- e.rank: RLE
- d.bldg: gzip Each department is distinct, so RLE won’t help here. We didn’t say anything about the range of bldg numbers, so delta encoding won’t help either.
- e.sal: gzip, or delta Delta might be a good idea as we have explicitly limited the range of salaries. Gzip will probably still be better.
- d.dno: delta When sorted on dno, the differences should be small (assuming these are monotonically increasing integer keys.), and will likely do better than gzip. A quick test shows that gzip on a file of monotonically increasing 4 byte ints (in binary format) yields 2.8x compression. Delta should approach 4x on this file.
- e.dno: RLE
II Cost Estimates

Suppose you have the following “equi-height” histogram on bank account balances in a database consisting of a table of bank accounts and their balances, as well as customers and their names and addresses:

Each bucket contains 100/6 = 16.7% of the records

<table>
<thead>
<tr>
<th>Balance</th>
<th>0</th>
<th>20K</th>
<th>40K</th>
<th>50K</th>
<th>65K</th>
<th>70K</th>
<th>100K</th>
</tr>
</thead>
</table>

3. [6 points]: Consider the predicate “balance > $50,000 and balance < $75,000”. Given the histogram above, what would Postgres estimate the selectivity $F$ of this predicate to be?

(Write your answer in the space below.)

$F = 0.167 \times (2 + 5/30)$

Your database has 10,000 bank accounts. You run the query:

```sql
SELECT COUNT(*)
FROM accounts, customers
WHERE balance > $50,000 and balance < $75,000
AND customers.state = 'MA'
AND accounts.custid = customers.custid
```

When you run the query, you find the COUNT it produces is just $1 \times 10,000 \times F$, where $F$ is the selectivity Postgres estimated in the previous question.

4. [5 points]:

In one sentence or less, provide one possible reason why the database might produce an answer that is so much less than expected, assuming that the histogram accurately reflects the true distribution of balances in the database:

(Write your answer in the space below.)

We accepted a range of answers due to the odd wording of the question (given that Massachusetts is a small state, and there are 50 states, it actually seems to have MORE people in this range than would be expected.) We accepted answers that observed that the histogram is built across all states but that the query is on MA, and the distribution of balances in MA could be very different than in all states.

Name:
III Indexing

You have a retail database consisting of a set of orders, each placed at one store, and each consisting of products. The table `lineitem` stores the list of products sold in each order, including the price paid per item and the quantity of each item purchased.

Consider the following query over this database:

```sql
SELECT orders.storeid, SUM(price * quantity)
FROM lineitem, orders, stores
WHERE stores.region = 'EAST'
AND lineitem.productId = 12345
AND lineitem.orderno = orders.orderno
AND orders.storeid = stores.storeid
GROUP BY orders.storeid
```

Here `storeid` is the primary key of `stores` and `orderno` is the primary key of `orders`. Suppose that there are 1,000 distinct products, 100,000 orders, 100 stores, 1,000,000 lineitems and 10 regions, and that products are equally popular, orders have uniform size, and stores are uniformly distributed across regions.

Your system has sufficient memory to store 50,000 records of any table in memory at a time and can read 10,000 records per second from disk. Seeks take 10 ms. Assume CPU costs are small but consider them when breaking ties if I/O costs are identical.

The database has access to the following join algorithms: grace hash, in-memory hash (when applicable), nested loops, sort merge, and index-nested loops. The system does not consider cross products.

5. [10 points]: Suppose you have no indexes.

In the query plan below, write the names of the tables (to specify the join order), and your choice of join algorithm for each join. Assume filters are pushed down as far as possible. For nested-loops joins, the leftmost join is the outer relation. For hash joins indicate which relation you would build the hash table on.

```
Name:
```
Explanation:

We can push the filters `lineitem.productId = 12345` and `stores.region = 'EAST'` down into the sequential scans of the lineitem and stores tables. Because we assume uniform distributions, these filters produce 1,000 lineitems and 10 stores. We can store up to 50,000 records in memory, so both of these filtered tables will fit in memory. This indicates that orders (the only table that can’t fit into memory after filtering) should be the outer relation for any join, and therefore placed in the lower left-hand slot.

Next we need to determine which of the remaining two tables to join with orders first. Many students chose to join with stores first since stores is the smaller table. However, the productld filter on lineitem is much more selective (selectivity = 0.001), and since we assume uniform distribution, this will produce a smaller join output. Joining lineitem with orders produces 1000 tuples, while joining lineitem with stores produces 10,000 tuples. Although both plans have the same I/O cost, the correct plan (with lineitem in the lower right) has a lower CPU cost.

Since both lineitem and stores can fit in memory after filtering, we can perform an in-memory hash join on both. We also accepted nested loops for the upper join with stores, since nested loops and in-memory hash will likely both perform well with only 10 records in the inner relation.

6. [8 points]: What B+Trees would you recommend the system create if this is the only query the system has to run? For each B+Tree, specify the tables/attributes it is on, and whether or not it is clustered. Assume you can have at most one clustered index per table (otherwise there are no constraints on how many indexes you can create.)

- lineitem: clustered index on productId
- stores: no index
- orders: clustered index on storeid

We did not take off points for additional indexes besides these, since we did not specify any index maintenance cost. See the next problem for an explanation of why these are the best indexes.

7. [10 points]: Given the indexes you created above, label the diagram below with the names of the tables, your choice of access method for each table, and your choice of join algorithm for each join. Assume filters are pushed down as far as possible (and into indexes when possible). For nested-loops joins, the leftmost join is the outer relation. For hash joins indicate which relation you would build the hash table on.

Name:
Explanation:

Access methods:
The clustered index on lineitem.productId is essential. Without it, a sequential scan of lineitem would take 100 seconds. With the index, we can quickly seek to the correct place in the heap file and sequentially scan the 1000 lineitems containing productId = 12345. We don’t know the height of the B-tree, but we can assume at least one seek to find the correct index page and one seek to find the correct heap file page. This gives us \((2 \times 10 \text{ ms}) + (1000 \text{ tuples} / 10,000 \text{ tuples/sec}) = 120 \text{ ms})\.

The stores table is small enough that it can be sequentially scanned in \(10 \text{ ms} + (100 \text{ tuples} / 10,000 \text{ tuples/sec}) = 20 \text{ ms})\). We also accepted answers in which students performed a clustered index scan on stores.region, since that would be faster if the index pages were cached. It is slightly slower \((22 \text{ ms})\) if we stick with the assumption of one seek to find the correct index page.

The correct access method for orders is a bit trickier. Since there is no filter on orders, we rely on the join algorithm to determine the correct access method. Without an index, it would take \(10 \text{ ms} + (100,000 \text{ tuples} / 10,000 \text{ tuples/sec}) = 10.01 \text{ seconds})\) to scan the table. As we will see, we can get this down to 1.2 seconds with a clever join strategy.

Join algorithms:
As mentioned above, the join algorithms need to be optimized to limit the time required to scan the orders table. The only join we have available that takes advantage of indexes is the index nested loops join, with orders as the inner relation. To determine the outer relation, let’s examine both possibilities: If we join with lineitem, we would create a clustered index on orders.orderno. The filtered lineitem table has 1000 tuples, which are likely from nearly 1000 different orders. This means we’d have to seek into the orders index and heap file 1000 times, for a total of \((1000 \times 20 \text{ ms}) + (1000 \text{ tuples} / 10,000 \text{ tuples/sec}) = 20.1 \text{ seconds})\). This is worse than the sequential scan, so obviously a bad choice. If we join with stores, however, we would create a clustered index on orders.storeid. The filtered stores table only has 10 stores, so we only need to seek into the orders index and heap file 10 times. We’ll end up scanning more orders, but overall it’s a win: \((10 \times 20 \text{ ms}) + (10,000 \text{ tuples} / 10,000 \text{ tuples/sec}) = 1.2 \text{ seconds})\).

In order to get this performance, we need to do the index nested loops join between stores and orders as the first join. The second join with lineitem can be a simple in-memory hash join since we’ve already filtered lineitem on productId, and 1000 tuples easily fit in memory.

In total, our query plan takes \(120 \text{ ms} + 20 \text{ ms} + 1.2 \text{ seconds} = 1.34 \text{ seconds})\.

Name:
IV Entity and Schema Design

Suppose you are designing a schema to record information about the intramural (IM) team sports at MIT, e.g. football, basketball, hockey, etc. Your database needs to record the following information:

- For each student, his/her student id (only MIT students for this problem), name, and address. A student may join different teams.
- For each team, its name, captain, ranking, and team members (including the captain).
- For each game, its host team, guest team, date, and score.

8. [10 points]: Draw an entity relationship diagram for this database. Please draw entities as squares, attributes as ovals, and denote relationships as diamonds between pairs of entities. Label each edge with a “1” or an “N” to indicate whether the entity on the other side of the relationship connects to 1 or N entities on this side of the relationship. For example, the following would indicate that each person lives in 1 city, and multiple people live in each city:

```
Person               City
| N   | 1 |
```

Give each entity, relationship, and attribute a name.

(Draw your diagram in the space below)

```
Students
<table>
<thead>
<tr>
<th>sid</th>
<th>name</th>
<th>address</th>
<th>isCaptain</th>
<th>membership</th>
</tr>
</thead>
</table>

Teams
<table>
<thead>
<tr>
<th>tid</th>
<th>name</th>
<th>ranking</th>
<th>host</th>
<th>guest</th>
</tr>
</thead>
</table>

Games
<table>
<thead>
<tr>
<th>gid</th>
<th>date</th>
<th>score</th>
</tr>
</thead>
</table>
```

Name:
9. [6 points]: Use your ER diagram to determine a relational schema for this database. For each table, use the form:

    TablenameN (field1-name, ..., fieldn-name)

to denote its schema. If you wish, you can create an additional field for each table to serve as a unique identifier. Underline the field(s) that are the primary key of each table.

Students(sid, name, address)

Teams(tid, name, ranking, captain references Students.sid)

Membership(sid references Students.sid, tid references Teams.tid)

Games(gid, host references Teams.tid, guest references Teams.tid, date, score)

10. [6 points]:

Suppose every team hosts at most one game a day, which yields the functional dependency (host team, date) → (guest team, score). How would you modify/decompose your schema to account for this dependency?

(Write your answer in the space below.)

We could drop gid for table Games and create primary key (host, date):

Games(host references Teams.tid, guest references Teams.tid, date, score)
V Query Planning

Suppose you are querying the mimic2 database to compare healthcare outcomes of patients who were cared for in a single unit during their ICU stay versus those who switched care units.

You run the following query to count the number of chart events for each class of patients:

```
SELECT count(*), 'moved' AS moved_status
FROM chartevents a, (SELECT icustay_id FROM icustayevents
    WHERE first_careunit <> last_careunit) AS moved
WHERE a.icustay_id = moved.icustay_id
UNION ALL
SELECT count(*), 'not moved' AS moved_status
FROM chartevents b, (SELECT icustay_id FROM icustayevents
    WHERE first_careunit = last_careunit) AS not_moved
WHERE b.icustay_id = not_moved.icustay_id;
```

This produces the following output:

```
          count    | moved_status
--------------------------------
         1942840 | moved
          7407892 | not moved
(2 rows)
```

Time: 4398.751 ms
To understand why this query took almost 5 seconds, you run EXPLAIN ANALYZE, and get the following output (some of the output has been omitted for clarity):

```
QUERY PLAN

-> Aggregate (cost=392078.93..392078.94 rows=1)
   (actual time=3068.571..3068.571 rows=1)
-> Hash Join (cost=53.35..368813.61 rows=9306127)
   (actual time=2.735..2895.545 rows=1942840)
   Hash Cond: (a.icustay_id = public.icustayevents.icustay_id)
-> Seq Scan on chartevents a (cost=0.00..227957.33 rows=9548444)
-> Hash (cost=33.73..33.73 rows=1570)
   (actual time=0.499..0.499 rows=148)
   Buckets: 1024 Batches: 1 Memory Usage: 6kB
   -> Seq Scan on icustayevents (cost=0.00..33.73 rows=1570)
   (actual time=0.021..0.457 rows=148)
   Filter: (first_careunit <> last_careunit)
   Rows Removed by Filter: 1430
-> Aggregate (cost=58817.74..58817.75 rows=1)
   (actual time=3663.863..3663.863 rows=1)
-> Nested Loop (cost=0.00..58699.19 rows=47420)
   (actual time=0.052..3065.650 rows=7407892)
   -> Seq Scan on icustayevents (cost=0.00..33.73 rows=8)
   (actual time=0.018..1.260 rows=1430)
   Filter: (first_careunit = last_careunit)
   Rows Removed by Filter: 148
-> Index Only Scan using chartevents_chartevents_o5 on chartevents b
   (cost=0.00..7267.40 rows=6578)
   (actual time=0.011..1.477 rows=5180)
   Index Cond: (icustay_id = public.icustayevents.icustay_id)
   Heap Fetches: 7407892
```

11. [8 points]: Notice that Postgres chooses very different query plans for the two aggregate queries. Using the EXPLAIN ANALYZE output as a guide, in a few sentences qualitatively explain the differences between the access methods and join algorithms Postgres chose for each aggregate query.

(Write your answer in the space below.)

Access Methods: The first aggregate query does a sequential scan on chartevents and icustayevents. The second aggregate query does a sequential scan on icustayevents and an index only scan on chartevents as part of an index nested loops join.

Join Algorithms: The first aggregate query does a hash join between icustayevents and chartevents, and builds the hash table on icustayevents. The second aggregate query does an index nested loops join between icustayevents and chartevents, with chartevents as the inner relation. Both queries use the join condition chartevents.icustay_id = icustayevents.icustay_id.

Name:
12. [10 points]: Based on the EXPLAIN output and what you know about query planning, which of the following statements must be true?

(Circle ‘T’ or ‘F’ for each choice.)

T  F  Postgres accurately estimates the cardinality of the two joins
FALSE Postgres is off by at least a factor of 10 in each case. In the first case, it estimates 9306127 rows, but there are actually only 1942840 rows. In the second case, it estimates 47420 rows, but there are actually 7407892 rows.

T  F  Postgres evaluates the “moved” subquery by doing a sequential scan on icustayevents once per chartevent
FALSE Postgres reads icustayevents once and builds a hash table on the filtered result. Then Postgres queries the hash table for each chartevent.

T  F  Postgres uses the chartevents table as the “inner” relation in both joins
FALSE In the first aggregate query, chartevents is the outer relation

T  F  The filter condition in the “moved” subquery is more selective (produces fewer rows) than the filter condition in the “not moved” subquery
TRUE The “moved” subquery produces 148 rows, while the “not moved” subquery produces 1430 rows.

T  F  The index nested loops join in the second aggregate query causes a seek into the chartevents heapfile for each iteration of the loop
FALSE Because Postgres uses an Index Only scan, it does not need to refer to the chartevents heap file.
In order to understand how the query would have performed with a different plan, we temporarily disable hash joins and merge joins, causing Postgres to change the plan it uses for the first aggregate query. When we run the query again, we see that it's over a second faster. EXPLAIN ANALYZE produces the following output (only the first aggregate query that changed is shown):

```
QUERY PLAN

-> Aggregate (cost=11320242.85..11320242.86 rows=1)
  (actual time=1764.008..1764.009 rows=1)
  -> Nested Loop (cost=0.00..11296977.54 rows=9306127)
    (actual time=0.080..1509.502 rows=1942840)
      -> Seq Scan on icustayevents (cost=0.00..33.73 rows=1570)
        (actual time=0.022..0.765 rows=148)
        Filter: (first_careunit <> last_careunit)
        Rows Removed by Filter: 1430
      Index Only Scan using chartevents_chartevents_o5 on chartevents a
        (cost=0.00..7129.73 rows=6578)
        (actual time=0.027..7.420 rows=13127)
        Index Cond: (icustay_id = icustayevents.icustay_id)
        Heap Fetches: 1942840
```

13. [10 points]: Briefly explain why the index nested loops join in this query plan is faster than the hash join in the previous plan. Give one reason why Postgres probably chose the wrong plan.

(Write your answer in the space below.)

The index nested loops (INL) join is faster because it avoids scanning the whole chartevents table. According to the EXPLAIN ANALYZE output, each index lookup takes 7.4 ms, so doing this 148 times takes just over a second. The overhead of scanning icustayevents and performing the join brings the total up to 1.5 seconds for the INL join. The hash join, however, requires scanning the whole chartevents table, which alone takes 1.4 seconds. Adding in the overhead of building the hash table on icustayevents and doing the join brings the total for this query up to 2.9 seconds. Put another way, the cost of the random I/O required for the index lookups in the INL join is less than cost of the sequential I/O required for the heap file scan in the hash join.

Postgres chose the wrong plan because it over-estimated the selectivity (over-estimated the number of rows returned) of the filter `first_careunit <> last_careunit`. As some students pointed out, this could be because Postgres assumed these two columns were independent, and therefore would generally not be the same.