Q1 Exam Instructions
0 Points

This exam is open book/note/internet, but you are expected to do the work on your own, without the help of other people (remote or local.) You are expected to complete the test in 50 minutes; it will cut you off after 60, but the extra 10 is for uploading handwritten answers; you are expected to complete your work in 50 minutes. Watch your time, and do not spend too long on any question. The times given are estimates - they are not enforced, but if you spend longer than the given time, you should probably go on and come back later.

You may ask questions in the office hours WebEx room during the hours it is staffed (hours were sent via email):
https://purdue.webex.com/purdue
/j.php?MTID=mde063787f89cf56c49f2a06c24adc1c2
(You should be able to call into this as well, +1-415-655-0003
Access code: 120 825 0068)

Clarifications will be posted at:
https://docs.google.com/document
/d/1AnfYJbZRMreHpsJFVuobcvlhAEiEXuHdpiNkppMyRw/edit?usp=sharing

Q1.1 Purdue Honor Code
0 Points

As a Boilermaker pursuing academic excellence, I pledge to be honest and true in all that I do. Accountable together - We are Purdue.

☐ I agree with the Purdue Honor Code.

☐ I do not agree with the Purdue Honor Code.
Q1.2 Answer upload method
0 Points

You may choose to do your exam either "online", or written on paper. If you choose the paper submission method, you will be able to download a PDF of this exam when you start the (Paper only submission) method. You can also have the online version open if you wish, but your start time will be based on when you first started either the online or paper version (if you open both, you may need to keep track yourself of when you started.)

☐ I will enter my answers in the text boxes and/or upload responses for each question. You must click "submit answer" before time expires. You can go back and change submitted answers (and resubmit) up until the time limit is reached, or you submit and view the completed exam.

☐ I will upload my entire answer set to Gradescope as a single PDF under "Midterm 2 (Paper only submission)" (just like you'd upload a regular written assignment, remember to mark your pages with the start of each question). Answers submitted here will not be viewed or graded.

Q1.3 Schema
0 Points

While you really won't need it to answer the questions, if you do find it useful you should assume the following schema.

student (ID, dept, lastname, firstname, asn1score, asn2score, asn3score )
employee (ID, dept, lastname, firstname, wage )
department(dept, deptname, address )

Assume that a person may be both an student and employee, in which case the IDs match (but it need not be the same department.) But a person can be just an employee or just a student, there is no foreign key relationship. *Italics* represent keys.
Q2 Indexes (5 minutes)
6 Points

For each of the following SQL queries, name or briefly describe a type of index that would be useful (or that not using an index would be better), and briefly describe why it is the right choice.

Q2.1 Query 1
2 Points

```sql
SELECT asn3score
FROM students
WHERE lastname = "Clifton"
```

This is an exact match query, where a dense hash index on students.lastname will take us to the correct record with one I/O to get the appropriate hash index block (assuming no overflow blocks), and one (or possibly more, depending on how many individuals are named "Clifton") to get the block(s) containing the record. Since we really can't avoid retrieving the records, our "search overhead" is one I/O.

An index clustered on lastname would be nice, since this would make it more likely all "Clifton"s would be in a single disk block, but this would likely conflict with the typical clustering on the primary key. Because of this, saying clustered alone doesn't get a point, but an explanation why you would want one does.

Q2.2 Query 2
2 Points

```sql
SELECT lastname
FROM students
WHERE asn3score >= 25
```
A B+ tree on students.asn3scores would be good, as it supports range search. Having a clustered index (sparse is good, but not necessary) would also be useful here, as we only need to find the first 25, then go from there.

With an unclustered index, if there are a lot of scores >= 25, then it is probably better to not use an index. In particular, if there are more tuples with scores >= 25 than blocks in the relation, it is likely to be faster to scan the relation than to use an index.

Q2.3 Query 3
2 Points

```
SELECT asn3_score, count(*)
FROM students
GROUP BY asn3score
```

This requires a full scan of the table, so an index won't help much. If there are a small number of distinct scores, we can simply keep the count for each, and update it as we make the pass. However, if we have a clustered index on asn3score, then we can make the pass, count, and output the tuple for each score/count as we go, which is more space-efficient, so it would help slightly (especially if there are too many distinct scores to store in memory.)

However, we can do one thing with a dense index. If we have a B+ tree, we can scan through the leaves, just counting the pointers to tuples (and not accessing the tuples). Similarly, with a hash index we can go through the hash buckets. This would require a full scan of the leaves of the B+ tree or all hash buckets (including overflow buckets), if this is smaller than the size of the table it would be faster.
Q3  B+ trees (7 minutes)
6 Points

Using the following B+-tree index on employee.wage:

Q3.1  Lookup (3 minutes)
3 Points

Given the SQL query:

```
SELECT *
FROM employees
WHERE wage>=10 and wage<=15
```

list which disk blocks will be access by the query, using the above index (e.g., D12, D17, ...)
We traverse the tree, then move along the leaves until we get to a value that is greater than 15. This gives us the blocks: D7 D14 D18 D6 D11 D13

Note that we have to look at D13, as there could be multiple occurrences of 15, one taking us into the next block.

A number of people tried to use the index to find the first and last blocks. You have to be careful, as going left from a node finds the leaf containing the first instance of the value you went left from, not necessarily the block with the last. If you go right, you'll find the first block with something greater (so going from D3 to D13, not D11, would work.) But in either case, it is going to require more disk accesses than simply traversing the leaves until you find a value greater than 15.

Q3.2 Delete (3 minutes)
3 Points

Describe or show the changes that will occur if we delete the record with wage 18. You can simply list the Disk block and the new context (e.g., D17: 27, 29) and describe changes to pointers.

Removing the entry from D2 makes the block underfull, so we need to merge blocks. Depending on how you implement, it could either add 19 to D13, or D16. Assume we add to D13, we would then need to update D3 to eliminate the "18" entry and the pointer to block D2, moving the 20 and pointer to D16 into their place and leaving the last entry in D3 blank.

Q4 Join algorithms (12 minutes)
9 Points

For each of the following three join algorithms, describe a situation
where 1) you expect it to perform better than the others, in terms of I/Os, and 2) you expect it to perform worse than others. "Situation" could include type of query, relation sizes, memory sizes, number of distinct attributes on join criteria, etc. If you think it could never be the best (or never worse than others), briefly describe why.

**Block Nested Loop Join:**
Performs better than others when:

Block Nested Loop works well when we can fit one relation entirely in memory, with room to load in a block at a time from the other relation, join everything in that block, and output the matches. This requires one pass through the small relation (to load it into memory), and one pass through the large relation (to load each block in and do the join.)

Performs worse than others when:

Block Nested Loop works poorly when we can't fit either relation entirely into memory, as we need to repeatedly read from one relation for each block of the other relation, resulting in a worst case of n*m (where n is the number of blocks in one relation, m in the other.) It can also be beat by an index join if the smaller relation is very small (fewer distinct values to join than the number of blocks in the larger relation.)

**Index Join:** (assume an index on one or both relations on the join attribute).
Performs better than others when:

Index join works best when there are few distinct values of the join key in one relation, and few matching tuples, so that the number of index lookups required, and the number of matches requiring us to retrieve the block containing the matched tuple, is small relative to the size of the indexed relation. This works best if the non-indexed relation is sorted or can be loaded entirely in memory, so we only need one index search for each distinct value of the join key. If the index fits in memory and there are few matching tuples, index join can be fast because
I/Os are only needed for matching tuples, potentially beating a full scan of the indexed table.

Performs worse than others when:

Index join works poorly whenever the number of matches (and corresponding block reads to get the matched tuples) is higher than the number of blocks in the indexed relation, as we would likely to better with a full table scan.

**Hash Join:**

Performs better than others when:

Hash join works well as long as the number of blocks in each table is less than the square of the memory size; otherwise hash join is unable to efficiently construct the hashed version of the table. If this constraint is met, hash join essentially requires two passes over each table ($2^{(m+n)}$ I/Os). If this constraint is not met, we need to use recursive partitioning, and a merge join may be better.

Performs worse than others when:

The conditions where for block nested loop and index join are good allow us to beat $2^{(m+n)}$ I/Os, and thus non-optimal for hash join. Also, if the relations are already sorted we can use merge-join in one pass ($m+n$ I/Os). Finally, even if our dataset is less than the square of the memory, if a particular key value occurs too frequently, we may not be able to fit a hash bucket into memory, requiring a block nested loop join for that bucket. But if this happens, the number of matching tuples, and thus the size of the output, will be so large that we're essentially looking at quadratic time for the output alone, so no method will be very efficient.
Given the following query:

```
SELECT lastname, address, wage
FROM employee, department
WHERE lastname = "Clifton" and deptname like "%Computer%"
```

Give two different relational algebra expressions (or query trees) that are equivalent to the SQL query.

```
\prod_{lastname, address, wage}(\sigma_{lastname="Clifton" \text{ and } deptname \text{ like } "%Computer%"}(employee, department))
```

But we can do better by pushing the selects lower, so the cross product doesn't produce so many tuples.

```
\prod_{lastname, wage}(\sigma_{lastname="Clifton"}(employee)) \times \\
\prod_{address}(\sigma_{deptname \text{ like } "%Computer%"}(department))
```

Which of the two do you think will run faster? Briefly describe why.

Note that this is a cross-product, not a join (the selection doesn't involve comparing attributes from both relations.) If you just used $\bowtie$, then this would be the natural join (employee.dept=department.dept), or if you specified the join condition e.dept=d.dept, then this isn't equivalent to the SQL given (even though it is semantically probably the intended query, best not to assume you know what the querier wants better than they do.) If you specified a $\theta$-join -

$\bowtie_{lastname="Clifton" \text{ and } deptname \text{ like } "%Computer%"}$ - it would be correct, as this is defined as a cross-product followed by a select, but confusing, since none of the algorithms we would use to make a join more efficient than a cross-product followed by a select would apply.

The second filters out many tuples (and attributes) before performing the cross-product, so the cross-product involves a
lot less data, making it faster. The first would result in
\( T(\text{employee}) \times T(\text{department}) \) tuples, probably requiring them to
be written back to disk (as it wouldn’t fit in memory), then doing
a nested loop, resulting in quadratic I/Os. In the second, the
selection on employee would likely give us a small enough
relation to fit in memory, which we could make the inner loop in
a nested loop, making only one pass through the employee
relation (if properly pipelined), giving linear I/Os. The second
also does duplicate elimination in the projection on a smaller
numbers of tuples, but this is a much smaller impact.

Q6 Cost estimation: Blocks Read (6 minutes)
6 Points

Given the query
\( \Pi_{\text{lastname}, \text{wage}} (employee \bowtie_{\text{employee.dept} = \text{department.dept}}
(\sigma_{\text{address} = \text{'West Lafayette'}} (deartment))) \), with cost estimates
given in the query tree below:

How many disk blocks would be needed for the results of the select
(the ??? in the picture)?

Since there are 10 distinct values for address, we would expect
one in 10 tuples to match, so the size would be 1/10 of the
original 25, or \( \lceil (2.5) \rceil = 3 \) blocks. (If you want to be pedantic,
you would calculate the expected tuples \( T(\text{select}) = \frac{1000}{10} \), and block size \( 1000 \times 100/25 = 4000 \), to get \( T(\text{select}) \times S(\text{select}) / \text{block size} = \frac{100 \times 100}{4000} = 2.5 \), giving the same result.)

Choose an appropriate join algorithm. Assume you have a small memory, your buffer can only hold 5 blocks. Briefly explain why you made this choice.

Since we have five blocks, we can keep the three in memory, so a block nested loop join with the selection from department as the inner relation would be able to keep the entire inner part in memory, turning it into one pass over the outer employee relation. This gives 125 blocks read total for the entire query - since this is the minimum needed to even see all the data.

If we had an index on employee, we could try an index join, but since \( T(\text{select}) \) would be 100, this leads to an expected 100 index lookups. If the entire index fit in three blocks, and we were lucky and several of our index lookups went to the block from employee already in memory from the last index lookup, it could potentially be less than 100 I/Os for the join. But if our index took more than three blocks, it would likely be much worse.

Q7 Cost estimation: Result set size (8 minutes)

6 Points

Given the query

\[
\sigma_{\text{address} = 'West Lafayette'}(\left( \text{student} \bowtie_{\text{student.d} = \text{employee.id}} \\text{employee} \right) \bowtie_{\text{employee.dept} = \text{department.dept}} \left( \Pi_{\text{dept.address}}(\text{department}) \right))
\]

(showed below, with relation size statistics):
Note that I haven’t given \( V(e, id) \) or \( V(s, id) \), but you should easily be able to figure these out (remember that id is a key for both relations.)

Estimate the result set sizes \( T(\text{join1}) \) and distinct values \( V(\text{join1, dept}) \) for the join of student and employee (the ??? in the picture).

The typical formula we would use is

\[
T(\text{join1}) = \frac{T(\text{student}) \cdot T(\text{employee})}{\max(V(\text{student}, \text{id}), V(\text{employee}, \text{id}))}
\]

But you aren’t given \( V(R, id) \). However, since id is a key, the number of distinct values must be the number of tuples, giving

\[
\frac{T(\text{student}) \cdot T(\text{employee})}{\max(T(\text{student}), T(\text{employee}))} = \frac{50000 \cdot 10000}{\max(50000, 10000)} = 10000.
\]

For \( V(\text{join1, dept}) \), since dept is only in employee, we use \( V(\text{join1, dept}) = V(\text{employee, dept}) = 100 \). We could be a bit more specific using the formula given in the book of \( V(\text{join1, dept}) = \min(V(\text{employee, dept}), T(\text{join1})) \), but this still gives 100.

We could change the order of operations (transform to an equivalent query). Give a transformation, and describe briefly if you think it should be faster or slower than the original and why. (You can use the relational algebra syntax rather than drawing the query tree, or even just describe the changed order of operations.)

Pushing the selection lower to give the query
would shrink our estimate of $T(\text{project})$ to 100.

There was an error in the question (well, two - grading took into multiple possible interpretations.) Assuming the right most relation, dept, was supposed to be "department", then dept is a key, so $V(d, \text{dept})$ would be $T(d)$, not 100. If you assume it is a key, then switching the order of the joins probably wouldn't make much difference, since $T(e)$ and $T(\text{join1})$ are the same. Although if we move the selection lower AND change the selection order, it would shrink the input to the join of student and (employee join dept), which would likely be the best result. If you used the values as written ($V(d, \text{dept}) = 100$, suggesting that each department may have multiple addresses), then switching the join order would slow things down, as the join of employee and department would be approximately 10 times the size of the employee relation, making the second join much larger.