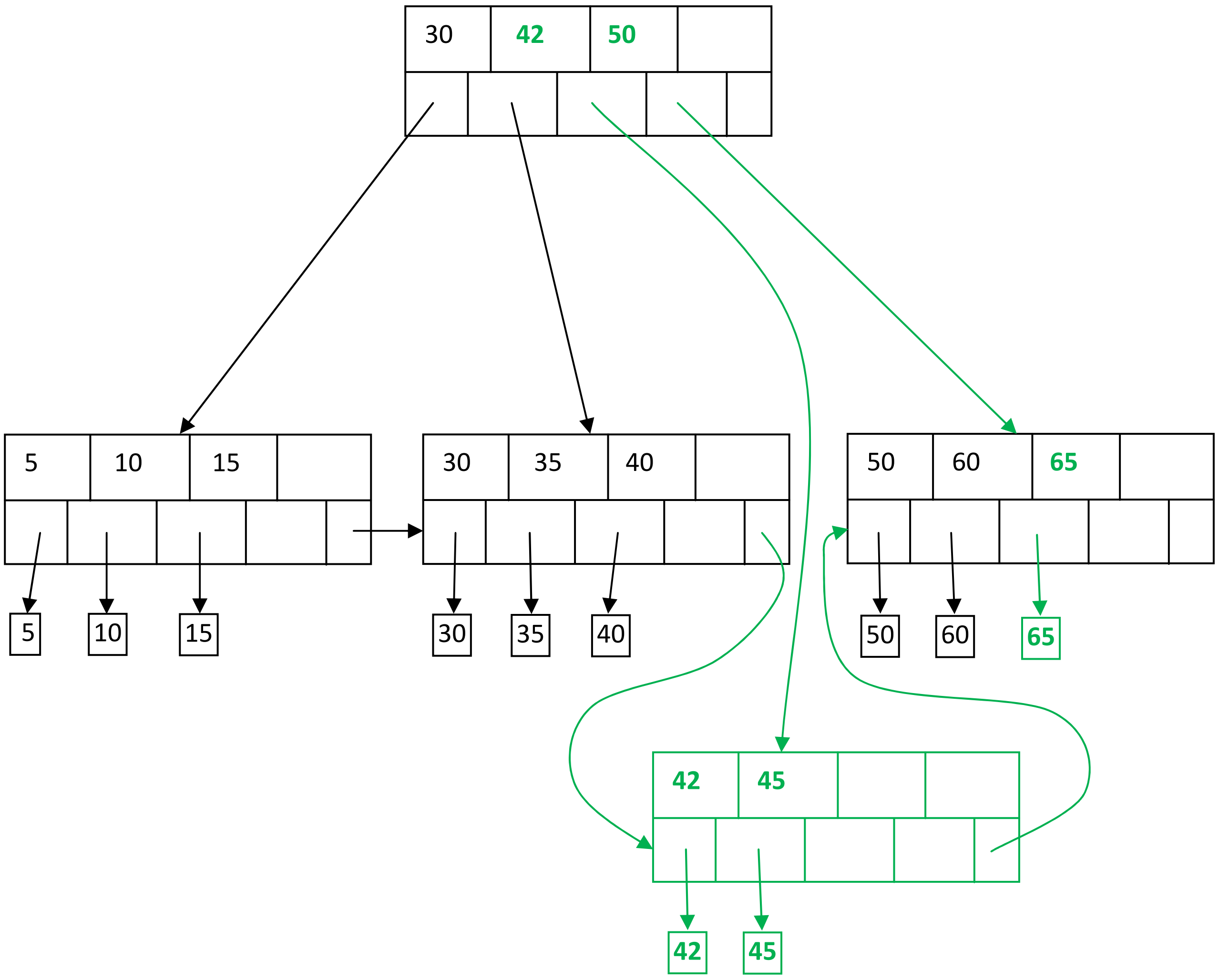
Question 1. B+ trees (15 points) Suppose we have the following B+ tree, where each index node has a maximum of 4 children.



Show how this B+ tree changes if the values 65 and 42 are inserted into the tree one after the other.

It would also be correct to split the middle node so the first one had two children (30 and 35) and the second had three (40, 42, and 45). That would change the middle index value in the root node to 40 instead of 42.

It is not correct to shuffle values from the middle node to either of its neighbors to avoid the split.

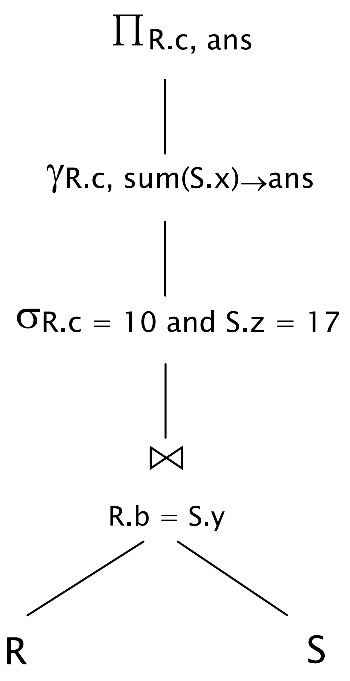
Question 2. Logical query plans (25 points) This problem and the next concern two relations R(a, b, c) and S(x, y, z) that have the following characteristics:

B(R) = 600	B(S) = 800
T(R) = 3000	T(S) = 4000
V(R, a) = 300	V(S, x) = 100
V(R, b) = 100	V(S, y) = 400
V(R, c) = 50	V(S, z) = 40

We also have M = 1000 (number of memory blocks).

Relation R has a clustered index on attribute a and an unclustered index on attribute c. Relation S has a clustered index on attribute x and an unclustered index on attribute z. All indices are B+ trees.

Now consider the following logical query plan for a query involving these two relations.


$\Pi_{R.c,\ ans}$
$\gamma_{R.c,\ sum(S.x) \rightarrow ans}$
$\sigma_{R.c = 10 \text{ and } S.z = 17}$
$\bowtie_{R.b = S.y}$
R
S

(Answer the questions about this query on the following page)

Question 2. (cont.) (a) Write a SQL query that is equivalent to the logical query plan on the previous page.

```
SELECT R.c, sum(S.x) AS ans
FROM R, S
WHERE R.b = S.y AND R.c = 10 AND S.z = 17
GROUP BY R.c
```

[One sharp student observed that the group-by hasn't got anything to do since there is only one value for R.c left after the select. However, γ is in the original relational algebra tree, so it should be here too. An earlier version of the question had R.c<10, but it was changed at the last minute to simplify the estimating, without catching that R.c=10 made the group-by basically irrelevant.]

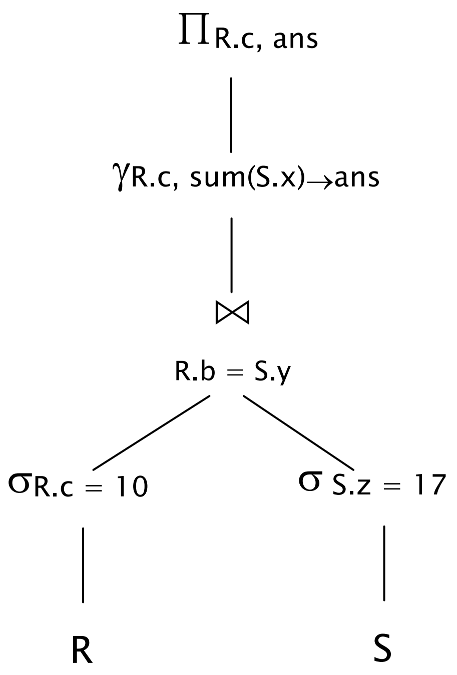
(b) Change or rearrange the original logical query plan to produce one that is equivalent (has the same final results), but which is estimated to be significantly faster, if possible. Recall that logical query optimization does not consider the final physical operators used to execute the query, but only things at the logical level, such as the sizes of relations and estimated sizes of intermediate results.

You should include a brief but specific explanation of how much you expect your changes to improve the speed of the query and why.

Draw your new query plan and write your explanation below. *If* you can clearly and easily show the changes on the original diagram, you may do so, otherwise draw a new diagram here.

In the original plan, the estimated size of the join result would be T(R)T(S)/max(V(R,b), V(S, y)) = 30,000 tuples, which then have to be passed through the select operation.

The most useful change would be to push the select operations down below the join. If that is done, the estimated result size of $\sigma_{R.c=10}$(R) is T(R)/V(R,c) = 60 and of $\sigma_{S.z=17}$(S) is T(S)/V(S,z) = 100. The estimated size of the join is then 60*100/400 or 15. The amount of work done by the group and project operations would remain the same as in the original plan.



[Grading note: many answers had calculations using B(R), B(S) or other variations, or talked about physical operator algorithms. But logical plans only deal with numbers of tuples and are independent of the physical plan. Also, although in this particular problem both relations had the same number of tuples per block, that is not normally the case and can't be assumed.]

Question 3. Physical query plans (25 points) This problem uses the same two relations and statistics as the previous one, but for a different operation not related to the previous query.

As before, the relations are R(a, b, c) and S(x, y, z). Here are the statistics again:

B(R) = 600	B(S) = 800
T(R) = 3000	T(S) = 4000
V(R, a) = 300	V(S, x) = 100
V(R, b) = 100	V(S, y) = 400
V(R, c) = 50	V(S, z) = 40

M = 1000 (number of memory blocks)

Also as before, relation R has a clustered index on attribute a and an unclustered index on attribute c. Relation S has a clustered index on attribute x and an unclustered index on attribute z. All indices are B+ trees.

Your job for this problem is to specify and justify a good physical plan for performing the join

$R \bowtie_{a=z} S$ (i.e., join R and S using the condition R.a = S.z)

Your answer should specify the physical join operator used (hash, nested loop, sort-merge, or other) and the access methods used to read relations R and S (sequential scan, index, etc.). Be sure to give essential details: i.e., if you use a hash join, which relations(s) are included in hash tables; if you use nested loops, which relation(s) are accessed in the inner and outer loops, etc.

Give the estimated cost of your solution in terms of number of disk I/O operations needed (you should ignore CPU time and the cost to read any index blocks).

You should give a brief justification why this is the best (cheapest) way to implement the join. You do not need to exhaustively analyze all the other possible join implementations and access methods, but you should give a brief discussion of why your solution is the preferred one compared to the other possibilities.

(Write your answer on the next page. You may remove this page for reference if you wish.)

Question 3. (cont) Give your solution and explanation here.

A simple hash join where both R and S are scanned sequentially will be as good as anything. The cost of that is B(R)+B(S). We have enough memory to store either R or S in a hash table, so there is particular advantage to picking either one if we view this query in isolation. From the system's standpoint, it would be better to hash R and then read blocks of S to perform the join. R is smaller, so storing it in the hash table will tie up less memory.

There is no advantage to using indices to read the blocks of the relations compared to a simple scan, and there can be a significant cost. We can scan the blocks of either relation at a cost of B(R) or B(S). If we look for tuples during the join using an index the costs will be higher, even with the clustered index, since we need to find matching tuples in one relation depending on the attribute values of the current tuple in the other, and those may be in any order.

A nested loop join is at least as expensive. We could hold all of one relation in memory and scan it repeatedly as we scan the other a block at a time, but the I/O cost will still be at least B(R)+B(S) and we will be repeatedly scanning the relation stored in memory. A naïve loop join where we read the blocks of the inner relation for each block of the outer one would cost B(R) + B(R)*B(S) or B(S) + B(S)*B(R), depending on which relation we scan in the outer loop, which would be more expensive.

A standard sort-merge join would be worse since we will need to sort one relation, write that to disk, sort the other, then read the first relation back a block at a time to join it to the sorted tuples of the other one. If we sort R first to minimize the number of blocks in the temporary file, the total cost of a sort-merge join would be 3B(R) + B(S). A naïve sort-merge join that writes both sorted relations to temporary files would be even worse, with a cost of 3B(R) + 3B(S).

If we were clever, we could do a sort-merge join where we sort S in memory at the cost B(S), then read the corresponding tuples of R using the clustered index on a at a cost of B(R) for a total cost of B(S) + B(R). But this has the same I/O costs as the hash join, and requires an extra sort operation.

[Grading notes: Answers were not expected to be this detailed or long-winded, but did need to pick the right algorithm, analyze the cost, and give at least a general argument about why this was the best choice.]

[Several answers reasoned that since B(R)/V(R,a) = 600/300 = 2, then the cost of reading R was 2. That number is an estimate for reading all of the tuples in R with a particular value for a, since R has a clustered index on a, but as we scan S there will be different values of S.z that we need to match. It is true that since V(S,z)=40, we can estimate that only 40 distinct values of R.a will be needed, which implies we might be able to read as few as 80 blocks of R. But then we would need to store those tuples in some sort of hash table so we can find them again as we read additional tuples in S that match. There may be some database systems that perform an optimization like this, but it is not one of the standard join algorithms, and nobody who estimated the cost of reading R as 2 did this kind of analysis to justify their approach.]

Question 4. XML (15 points) It's never too early to think about the holidays – at least not if you are in the toy business. Santa's elves are using XML to keep track of the toys in Santa's sack this year. The DTD for their data is the following:

```
<!DOCTYPE sack [
<!ELEMENT sack (toy)* >
<!ELEMENT toy (name, color*, age?) >
<!ELEMENT name (#PCDATA) >
<!ELEMENT color (#PCDATA) >
<!ELEMENT age (#PCDATA) >
]>
```

Although it is not specified in the DTD itself, the values for age will be one of "baby", "child", "teen", "adult", or "everyone", if age is present in an entry. If a toy is available in multiple colors, it will have multiple color attributes in its entry. Here is a short example of the kind of data that we might find in Santa's database:

```
<sack>
    <toy>
        <name>firetruck</name>
        <color>red</color>
        <age>child</age>
    </toy>
    <toy>
        <name>rattle</name>
        <color>pink</color>
        <color>blue</color>
        <age>baby</age>
    </toy>
</sack>
```

On the next page, write XPath or XQuery expressions as requested to perform the desired tasks.

(You may detach this page for reference if you wish.)

Question 4. (cont). Use the XML schema on the previous page to answer these questions. If it matters, you can assume that the data is stored in a file named "sack.xml".

(a) Write an XPath expression (not a more general XQuery) that returns the names of all toys that are available in the color purple (i.e., have at least one color entity with the value "purple").

```
/sack/toy[color='purple']/name/text()
```

[The "text()" is needed to return just the name without the surrounding tags, but we didn't count off if that was omitted since the question wasn't quite explicit about it.]

(b) Write an XQuery expression that reformats a valid XML document specified by the original DTD to give a list of toy names grouped by age. The resulting document should include all of the toys in the original document and should match this DTD:

```
<!DOCTYPE agelist [
<!ELEMENT agelist (group)* >
<!ELEMENT group (age, name*) >
<!ELEMENT name (#PCDATA) >
<!ELEMENT age (#PCDATA) >
]>
```

```
<agelist> {
    for $d in document("sack.xml")/sack
    for $a in distinct-values($d/toy/age/text())
    return <group>
                <age> { $a } </age>
                { for $n in $d/toy[age=$a]/name
                    return  { $n }
                }
            </group>
    }
</agelist>
```

[This question should have been explicit about how to handle <toy>s that did not have the optional <age> entity. We gave credit for answers that worked on <toy>s with <age> entities and did not worry about what to do if <age> was not present.]

Question 5. Map-Reduce (20 points) For each of the following problems describe how you would solve it using map-reduce. You should explain how the input is mapped into (key, value) pairs by the map stage, i.e., specify what is the key and what is the associated value in each pair, and, if needed, how the key(s) and value(s) are computed. Then you should explain how the (key, value) pairs produced by the map stage are processed by the reduce stage to get the final answer(s). If the job cannot be done in a single map-reduce pass, describe how it would be structured into two or more map-reduce jobs with the output of the first job becoming input to the next one(s).

You should just describe your solution algorithm. You should not translate the solution into Pig Latin or any other detailed programming language.

(a) The input is a list of housing data where each input record contains information about a single house: (address, city, state, zip, value). The output should be the average house value in each zip code.

Map output: from each individual input record, key = zip; value = value

Reduce output: key = zip; value = average of all input values with the same zip.

(b) [Same as the last part of project 3, only this time using map-reduce instead of SQL] The input contains two lists. One list gives voter information for every registered voter: (voter-id, name, age, zip). The other list gives disease information: (zip, age, disease). For each unique pair of age and zip values, the output should give a list of names and a list of diseases for people in that zip code with that age. If a particular age/zip pair appears in one input list but not the other, then that age/zip pair can appear in the output with an empty list of names or diseases, or you can omit it from the output entirely, depending on which is easier.

(Hint: the keys in a map/reduce step do not need to be single atomic values.)

Several answers used multiple jobs for this, but it can be done in one.

Map output: key = (age, zip) from each input record from either file; value = either a name or disease with some information to distinguish which it is, or a pair of lists one of which is empty and another that contains a single name or a disease as appropriate.

Reduce output: key = (age, zip); value = lists of all names and diseases concatenated from all input records with the same (age, zip) key.

[Some of the answers to these questions were amazingly verbose… .]