CS 186 Exam Prep Section 3
Iterators, Joins and Query Optimization

Agenda

I. 5:10-5:30 -- Mini-lecture
II. 5:30-6:30 -- Worksheet
III. 6:30-7 -- Go over answers

Worksheet can be at @376 on Piazza. Slides can be found on the same Piazza post.

Anonymous Feedback Form: https://tinyurl.com/CS186ExamPrep3FA20

Iterators and Joins

Iterators

- Used by the operators (joins, table scans, etc.) that compose query plans
- Iterators output the records resulting from the operator's operation, which are then fed as input to the next iterator.
- Because iterators can feed into each other without writing to disk, **exclude write cost**

Notation

- [R] - number of pages in R
- p_R - number of records per page in R
- |R| - number of records in R

Joins

- Simple / Page / Block Nested Loop Joins:
  - (all pages of left table)
  - + (number of passes of right table) * (all pages of right table)
- Number of passes:
  - Simple: one per left row
  - Page: one per left page
  - Block: one per left block

Notation

- [R] - number of pages in R
- p_R - number of records per page in R
- |R| - number of records in R

Joins

- Simple Nested Loop Join: [R]+|R|[S]
- Page Nested Loop Join: [R] + [R][S]
- Block Nested Loop Join: $[R] + \lceil \frac{[R]}{B-2} \rceil [S]$ where B is the number of available buffer pages

Notation

- [R] - number of pages in R
- p_R - number of records per page in R
- |R| - number of records in R

Joins

- Index Nested Loop Join: [R] + |R| * cost to find matching S tuples
  - (all pages of left table)
  - + (number of right index lookups) * (cost of right index lookup)

- Cost to find matching S tuples:
  - Alternative 1: just cost to traverse root to leaf + read all the leaves with matching tuples
  - Alternative 2/3: cost of retrieving RIDs (similar to Alternative 1) + cost to fetch actual records
    - 1 I/O per **page** if clustered, 1 I/O per **tuple** if not

Notation

- [R] - number of pages in R
- p_R - number of records per page in R
- |R| - number of records in R

Joins

- Sort Merge Join:
  - cost to sort R using external sorting
  - + cost to sort S using external sorting
  - + [R] + [S]
  - Note that, if a relation is already sorted, we can exclude that cost

Notation

- [R] - number of pages in R
- p_R - number of records per page in R
- |R| - number of records in R

Joins

- Sort Merge Join optimization: combine **last sort pass** with **merging**

Normally:

- Last sort pass:
  - Load runs R_1, R_2, R_3 into buffers, merge into run R, stream (write) R to disk
  - Load runs S_1,S_2, S_3 into buffers, merge into run S, stream (write) S to disk
- Merging:
  - Load run R and run S into buffers, merge into R⋈S

Notation
- [R] - number of pages in R
- p_R - number of records per page in R
- |R| - number of records in R

Joins

- Sort Merge Join optimization: combine **last sort pass** with **merging**

Sort-merge optimization:

- Last sort pass:
  - Load runs R_1, R_2, R_3 into buffers~~, merge into run R, stream (write) R to disk~~
  - Load runs S_1,S_2, S_3 into buffers~~, merge into run S, stream (write) S to disk~~
- Merging:
  - ~~Load run R and run S into buffers,~~ merge into R⋈S

Note that in this example, previously we needed only 3 input buffers, but the optimized version needed 6 input buffers!

In general, this optimization is only possible if you happen to have enough buffers to stream BOTH last runs in memory. You can also do a partial version where you finish sorting one table normally, then do the join with the runs of the unmerged table and the one run of the merged table.

You save 2 *([R] + [S]) by doing this optimization. The partial version saves either 2 * [R] or 2 * [S], depending on which table you wait to merge.

Notation

- [R] - number of pages in R
- p_R - number of records per page in R
- |R| - number of records in R

Joins

- Grace Hash Join: similar to external hash, but…
  - Partitioning phase: Partition R into B-1 buckets and also S into B-1 buckets
  - Recursively partition pairs of R and S partitions until one partition in a pair fits in B-2 pages
  - Joining phase: for each pair of partitions where at least one is at most B-2 pages,
    - load smaller side (e.g. R) into memory, and make a hash table
    - Stream in pages of S -> match against hash table -> stream out matches
- Cost: Depends on the construction of the tables. It's similar to external hashing, but your parameters for stopping are different

Query Optimization

Query Optimization

```
SELECT travelers.name, cities.name
FROM travelers left outer join cities on city_id = dest_id
WHERE cities.name == 'Berkeley'
ORDER BY cities.name;
```

- Many different orders to perform all these operations
- We use the System R optimizer (aka Selinger optimizer)
- Plan space: **only left-deep trees (important!), avoid cartesian products**
- Cost estimation: We'll only use I/O cost for this class (exclude CPU)
- Search algorithm: dynamic programming

Selectivity Estimation

- To estimate the cost of a query, add up the estimated costs of each operator in the query
  - Need to know the size of the **intermediate relations** (generated from one operator and passed into another)
    - Need to know the **selectivity** of predicates - what % of tuples are selected by a predicate
- These are all estimates: if we don't know, we make up a value for it (selectivity = 1/10)
- System R assume uniform and indep. distribution of values

Selectivity Estimation - Equalities

Predicate	Selectivity	Assumption
c = v	1 / (number of distinct values of c in index)	We know |c|.
c = v	1 / 10	We don't know |c|.
c1 = c2	1 / MAX(number of distinct values of c1, number of distinct values of c2)	We know |c1| and |c2|.
c1 = c2	1 / (number of distinct values of ci)	We know |ci| but not |other column|.
c1 = c2	1 / 10	We don't know |c1| or |c2|.

Included for completeness - don't memorize, just put on your reference sheet

|column| = the number of distinct values for the column

Note: If you have an index on the column, you can assume you know |column|, max(c), and min(c)

Selectivity Estimation - Inequalities on Integers

Predicate	Selectivity	Assumption
c > v	(high key - v) / (high key - low key + 1)	We know max(c) and min(c).
c > v	1 / 10	We don't know max(c) and min(c).
c >= v	(high key - v) / (high key - low key + 1) + (1 / number of distinct values of c)	We know max(c) and min(c).
c >= v	1 / 10	We don't know max(c) and min(c).

Selectivity Estimation - Inequalities on Integers

Predicate	Selectivity	Assumption
c < v	(v - low key) / (high key - low key + 1)	We know max(c) and min(c).
c < v	1 / 10	We don’t know max(c) and min(c).
c <= v	(v - low key) / (high key - low key + 1) + (1 / number of distinct values of c)	We know max(c) and min(c).
c <= v	1 / 10	We don’t know max(c) and min(c).

Selectivity Estimation - Connectives

Predicate	Selectivity	Assumption
p1 AND p2	S(p1)S(p2)	Independent predicates
p1 OR p2	S(p1) + S(p2) - S(p1)S(p2)	
NOT p	1 - S(p)	

Query Optimization - Selinger

- Pass 1: find minimum cost access method for each (relation, interesting order)
  - Index scan, full table scans
- Pass i (for $1 < i \leq n$): take in list of optimal plans for (i - 1 relations, interesting order) from Pass i-1, and compute minimum cost plan for (i relations, interesting orders) (every size i subset of the n relations)

Query Optimization Selectivity Order

- Apply single table predicates in the first pass such as `WHERE student.age > 3000000000`
- Multiple table predicates such as `Student.SID = Cs186student.SID` take place in subsequent passes

Query Optimization

- For n relations joined, perform n passes
  - on the i-th path, output only the best plan for joining any i of the n relations
  - Also keep around plans that have higher cost but have an **interesting order**
- **This along with only considering left-deep plans forms the crux of most QO questions**

Query Optimization - Interesting Orders

- **Interesting orders** are orderings on intermediate relations that may help reduce the cost of later **joins**
  - ORDER BY attributes
  - GROUP BY attributes
  - *downstream* join attributes
    - For instance, sort merge join will produce a relation that can help with an ORDER BY clause

Worksheet

Solutions

Joins 1

Joins 1

```
CREATE TABLE Students (
    student_id INTEGER PRIMARY KEY,
    ...
);

CREATE TABLE AssignmentSubmissions(
    assignment_number INTEGER,
    student_id INTEGER REFERENCES Students(student_id),
    ...
);

SELECT *
FROM Students, AssignmentSubmissions
WHERE Students.student_id = AssignmentSubmissions.student_id;
```

We also have:

- `Students` has $[S] = 20$ pages, with $p_S = 200$ records per page
- `AssignmentSubmissions` has $[A] = 40$ pages, with $p_A = 250$ records per page

Question 1

What is the I/O cost of a simple nested loop join for Students ⋈ AssignmentSubmissions?

Answer: **160,020 I/Os**.

The formula for a simple nested loop join is [S] +|S|·[A].

Plugging in the numbers gives us 20 + (20·200)·40 = 160,020 I/Os

Question 2

What is the I/O cost of a simple nested loop join for AssignmentSubmissions ⋈ Students?

Answer: **200,040 I/Os.**

The formula for a simple nested loop join is [A] +|A|·[S].

Plugging in the numbers gives us 40 + (40·250)·20 = 200,040 I/Os

Question 3

What is the I/O cost of a block nested loop join for Students ⋈ AssignmentSubmissions? Assume our buffer size is B= 12 pages.

Answer: **100 I/Os**.

First, we can calculate our block size: B−2 = 10.

Since Students is our left table, we calculate the number of blocks of Students :[S]/(B−2) = 20/10 = 2.

Thus the final cost is [S] plus 2 passes through all of [A], or 20 + 2·40 = 100 I/Os.

Question 4

What is the I/O cost of a block nested loop join for AssignmentSubmissions ⋈ Students? Assume our buffer size is B= 12 pages.

Answer: **120 I/Os**.

As before, we can calculate our block size: B−2 = 10.

Since AssignmentSubmissions is our left table, we calculate the number of blocks: [A]/(B−2) = 40/10 = 4.

Thus the final cost is [A] plus 4 passes through all of [S], or 40 + 4·20 = 120 I/Os.

Question 5

What is the I/O cost of an Index-Nested Loop Join for Students ⋈ AssignmentSubmissions? Assume we have a clustered alternative 2 index on AssignmentSubmissions.studentid, in the form of a height 2 B+ tree. Assume that index node and leaf pages are not cached; all hits are on the sameleaf page; and all hits are also on the same data page.

Answer: **16,020 I/Os**.

The formula is [S] +|S|·(cost of index lookup).

The cost of index lookup is 3 I/Os to access the leaf, and 1 I/O to access the data page for all matching records.

So the total cost is 20 + 4000·4 = 16,020 I/Os.

Question 6

Now assume we have an unclustered alternative 2 index on AssignmentSubmissions.studentid, in the form of a height 2 B+ tree. Assume that index node pages and leaf pages are never cached, and we only need to read the relevant leaf page once for each record of Students, and all hits are on the same leaf page. What is the I/O cost of an Index-Nested Loop Join for Students ⋈ AssignmentSubmissions?

Answer:**22,020 I/Os**.

The formula is [S] +|S|·⟨cost of index lookup. This time though, the cost of index lookup is 3 I/Os to access the leaf, and 1 I/O to access the data page for each matching record.

How many records match per key? We actually haven't told you! But, we do know that we will eventually have to access each record exactly once (since each AssignmentSubmission is foreign-keyed on a studentid) - so there will be |A|= 10,000 data page lookups, one for each row. So the total cost is 20 + 4000·3 + 10000 = 22,020 I/Os.

Question 7

What is the cost of an unoptimized sort-merge join for Students ⋈ AssignmentSubmissions? Assume we have B = 12 buffer pages.

Answer:**300 I/Os**.

The formula is⟨cost of sorting S⟩+⟨cost of sorting A⟩+ [S] + [A].

For sorting S: The first pass will make two runs, which is mergeable in one merge pass; thus, we need two passes; For sorting A: The first pass will make four runs, which is mergeable in one merge pass; thus, we need two passes.

Thus the total cost is (2·2[S]) + (2·2[A]) + [S] + [A] = 5([S] + [A]) = 5·60 = 300 I/Os.

Question 8

What is the cost of an optimized sort-merge join for Students ⋈ AssignmentSubmissions? Assume we have B = 12 buffer pages.

Answer: **180 I/Os**.

The difference from the above question is that we will skip the last write in the external sorting phase, and the initial read in the sort-merge phase. For this to be possible, all the runs of S and A in the last phase of external sorting should be able to fit into memory together.

From the previous question, we know there are 2 + 4 = 6 runs, which fits just fine in our buffer of 12 pages. Thus the total cost is 300−2[S]−2[A] = 300−120 = 180 I/Os.

Question 9

In the previous question, we had a buffer of B= 12 pages. If we shrank B enough, the answer we got might change. How small can the buffer B be without changing the I/O cost answer we got?

Answer: **9 buffer pages.**

The restriction for optimized sort-merge join is that the number of final runs of S and A can both fit in memory simultaneously. (i.e., the number of runs of S + the number of runs of A≤B - 1). We had 2 + 4 runs last time, which fit comfortably in 12−1 buffer pages (recall that one page is reserved for output).

What about B= 9? Now we have 3 runs for S and 5 runs forA, which just exactly fits in 9−1 buffer pages.Since 9 buffer pages fits perfectly, any smaller would force more merge passes and thus more I/Os.

Question 10

What is the I/O cost of Grace Hash Join on these tables? Assume we have a buffer of B = 6 pages
Answer: 180 I/Os

For Grace Hash Join, we have to walk through what the partition sizes are like for each phase, one phase at a time. In the partitioning phase, we will proceed as in external hashing. We will load in 1 page a time and hash it into B−1 = 5 partitions. This means the 20 pages of S get split into 4 pages per partition, and the 40 pages of A get split into 8 pages per partition.

Do we need to recursively partition? No! Remember that the stopping condition is that any table's partition fits in B−2 = 4 buffer pages; the partitions of S satisfy this.In the hash joining phase, the I/O cost is simply the total number of pages across all partitions - we read all of these in exactly once. Thus the final I/O cost is 20 + 20 for partitioning S, 40 + 40 for partitioning A, and 20 + 40 for the hash join, for a total cost of 180 I/Os.

Joins 2

Joins 2

Consider a modified version of the baseball database that only stores information from the last 40 years.

```
CREATE TABLE Teams (
    team_id INTEGER PRIMARY KEY,
    team_name VARCHAR(20),
    year INTEGER,
    ...
);

CREATE TABLE Players(
    player_id INTEGER PRIMARY KEY,
    team_id INTEGER REFERENCES Teams(team_id),
    year INTEGER,
    ...
);
```

Each record in Teams represents a single team for a single year. Each record in Players represents a single player during a single year. We also have:

- `Teams` has $[T] = 30$ pages, with $p_T = 40$ records per page
- `Players` has $[P] = 300$ pages, with $p_P = 50$ records per page

Question 1

What is the I/O cost of a simple nested loop join for joining Teams ⋈ Players on Teams.team id = Players.team id?

Answer: **360,030 I/Os**.

The formula for a simple nested loop join is $[T] + |T| \cdot [P]$.

Plugging in the numbers gives us $30 + (30 \cdot 40) \cdot 300 = 360,030$ I/Os

Question 2

What is the I/O cost of a page nested loop join on the same query?

Answer: **9,030 I/Os**.

The formula for a simple nested loop join is [T] + [T] · [P].

Plugging in the numbers gives us 30 + 30 · 300 = 9,030 I/Os

Question 3

What is the I/O cost of a block nested loop join on the same query? Assume our buffer size is B = 10 pages.

Answer: **1,230 I/Os**.

The formula for a block nested loop join is [T] + ceil([T]/(B − 2)) · [P].

Plugging in the numbers gives us 30 + 4 · 300 = 1, 230 I/Os.

Question 4

Assume we have an unclustered index of height 1 on Teams.team id. What is the I/O cost of an index nested loop join on Teams.team id = Players.team id using this index? You can assume that every player only plays on one team each year.

Answer: **45,300 I/Os**.

If we are using an index on Teams.team id, this means that Teams should be the inner relation, because for each record in Players, we will search our index for the corresponding record in the Teams table. Thus the formula for the index nested loop join is [P] + |P|· cost to find matching records. Each search will cost 3 I/Os (2 to read the root + leaf, and 1 additional I/O to read a data page).

Plugging in the numbers gives us 300 + (300 · 50) · 3 = 45,300 I/Os.

Question 5

Now, assume we have a clustered index of height 2 on Players.player id and an clustered index of height 3 on Players.team id. If each team has 25 players each year, what is the lowest I/O cost of the join on Teams.team id = Players.team id using one of these indexes?

Answer: **6,030 I/Os**.

Although the index on Players.player id has a lower height, it is not useful for this join since the player id column is not part of the join. We must use the clustered index of height 3 on Players.team id. Since we are using an index on Players.team id, the Players table should be the inner relation. The formula will be [T] + |T|· cost to find matching records. Each search will cost 5 I/Os (4 to read down to the leaf level, and 1 additional I/O to read a data page since it is clustered).

Plugging in the numbers gives us 30 + (30 · 40) · 5 = 6,030 I/Os.

Question 6

Assume the index on Players.team id is actually unclustered. What is cost of the join on on Teams.team id = Players.team id using this index?

Answer: **34,830 I/Os**.

The equation will still be [T] + |T|· cost to find matching records; the only difference from the previous question is that the cost of searching matching records will be different, since it is unclustered.

Each search will cost 29 I/Os (4 to read down to the leaf level, and 25 additional I/Os to read data pages) since we must assume that the 25 player records for each team are on separate data pages.

Plugging in the numbers gives us 30 + (30 · 40) · 29 = 34, 830 I/Os.

Question 7

Consider a universe where there are no limits on team size and players get to time travel to play for whatever team they want, and 75% of players choose to travel to 2020 to play for the best baseball team, the San Diego Padres. In other words, imagine that 75% of the records in the Players table have the same team_id. What are the effects on the performance of the different join algorithms?

Simple, page, and block nested loop join would have the same performance as the original scenario.

Index nested loop joins could potentially be improved if we had enough buffer pages to keep the leaf node corresponding to the repeated team_id in memory.

Question 7

Consider a universe where there are no limits on team size and players get to time travel to play for whatever team they want, and 75% of players choose to travel to 2020 to play for the best baseball team, the San Diego Padres. In other words, imagine that 75% of the records in the Players table have the same team_id. What are the effects on the performance of the different join algorithms?

For sort merge join, many duplicate values generally tend to increase the number of I/Os, since iterators need to be reset. In this scenario, using the Players table as the outer relation during the merge phase could alleviate these concerns, since there are no duplicate team_id values in the Teams table.

Question 7

Consider a universe where there are no limits on team size and players get to time travel to play for whatever team they want, and 75% of players choose to travel to 2020 to play for the best baseball team, the San Diego Padres. In other words, imagine that 75% of the records in the Players table have the same team_id. What are the effects on the performance of the different join algorithms?

Grace hash join requires us to partition our data until at least one of our tables has chunks that are small enough to fit into memory. Since the Teams table does not have duplicate team_id values, there should not be significant differences in the number of I/Os compared to the original scenario.

Joins 3

Joins 3

In the following problems, we will be joining two tables: Students and AssignmentSubmissions on the key ‘student_id‘. However, we are dealing with a set of system constraints. Given a set of potential join algorithms from SNLJ, BNLJ, PNLJ, Hash Join, GHJ, SMJ, select the best option(s).

```
CREATE TABLE Students (
 student_id INTEGER PRIMARY KEY,
 ...
);
```

```
CREATE TABLE AssignmentSubmissions(
 assignment_number INTEGER,
 student_id INTEGER REFERENCES Students(student_id),
 ...
);
```

SELECT *FROM Students, AssignmentSubmissions WHERE Students.student_id = AssignmentSubmissions.student_id;

Unless otherwise explicitly stated, there are 600 pages of Students, 600 pages of AssignmentSubmissions, 60 records/page for each and 10 buffer pages. Assume all student_ids are unique between both tables. All parts are independent of each other

Question 1

Our program memory is extremely limited (not buffer memory)! As a result, we only have enough memory to store 1 hash function. What join algorithms work here provided it fits our system constraints)? Why?

Anything that's not GHJ or Hash Join. Hash Join requires one of the relations to fit in memory (B-2 pages). For GHJ, since we have only 1 hash function, we can not recursively partition. Remember for recursive partitioning, a new, independent hash function must be used in order to effectively re-partition the data.

Question 2

Now, we have very little buffer memory (only 3 pages). What join algorithms work here? How are the different join algorithms affected by the given constraint? Why?

BNLJ, PNLJ, and SNLJ would work. Given the limited buffer memory, BNLJ will reduce to PNLJ, since the size of the blocks will be limited to 1 page. In other words, the BNLJ and PNLJ will have the same IO cost. Note that the limited buffer memory does not impact SNLJ, since SNLJ simply takes each record in the first relation and searches for all matches in the second relation.

Naive Hash Join will not work, since neither of the relations can directly fit in B-2 pages. To fit in B-2 pages means that we can create a hash table for that relation.

GHJ would work with this constraint. We can repeatedly hash the two relations into B-1 buffers to create partitions that are $\leq B - 2$ pages big, which eventually allows us to fit the relations into memory and perform a Naive Hash Join. Due to the limited buffer memory, GHJ will require a significant number of partitioning passes, which in turn results in an increase in the overall IO cost.

SMJ would work with this constraint. Due to the limited buffer memory, SMJ would require a significant number of passes to sort a relation. This in turn will result in an increase in the overall IO cost.

Question 3

Now, assume that Students is only 1 page. What is the best join algorithm IO wise?

All join algorithms would work in this case, since our requirements are looser than than when Students had 600 pages. PNLJ/BNLJ/HJ/GHJ is the best IO wise.

SNLJ: [R] + p_r[R][S] = 1 + 60(1 * 600) = 36001 IOs
PNLJ: [R] + [R][S] = 1 + (1 * 600) = 601 IOs
BNLJ: [R] + ceil([R]/(B-2))[S] = 1 + 1 * 600 = 601 IOs
HJ/GHJ: [R] + [S] = 1 + 600 = 601 IOs
SMJ: Since our tables are not yet sorted, we'd incur some overhead to sort them, which would take a significant number of IOs.

Question 4

Now, assume that all student_ids in both tables are exactly the same (i.e. assume the primary key constraint does not hold). What join algorithms work here? How are the different join algorithms affected by the given constraint? Why?

GHJ and Hash Join would not work in this case. Since all student_ids are exactly the same, GHJ and Hash Join will be subject to extreme data skew (data hashing to the same bucket).

The nested loop joins (BNLJ, PNLJ, SNLJ) and SMJ will work in this case, however, it is worth considering what these joins will yield. Assuming that the joins are performed by matching student_ids, the nested loop joins (BNLJ, PNLJ, SNLJ) and SMJ will match every row from the left relation to all rows in the right relation.

Query Optimization 1

Query Optimization 1

```
SELECT Student.name, Company.open_roles, Application.referral
FROM Student, Application, Company
WHERE Student.sid = Application.sid
AND Application.cid = Company.cid
AND Student.semesters_completed > 6
AND (Student.major='EECS' OR Company.open_roles <= 50)
AND NOT Application.status = 'limbo'
ORDER BY Company.open_roles;
```

Table Schema	Records	Pages	Indices
CREATE TABLE Student (sid INTEGER PRIMARY KEY, name VARCHAR(32), major VARCHAR(64)), semesters_completed INTEGER)	25,000	500	• Index 1: Clustered(major). There are 130 unique majors • Index 2: Unclustered(semesters completed). There are 11 unique values in the range [0, 10]
CREATE TABLE Application (sid INTEGER REFERENCES Student, cid INTEGER REFERENCES Company, status TEXT, (sid, cid) PRIMARY KEY)	100,000	10,000	• Index 3: Clustered(cid, sid). • Given: status has 10 unique values
CREATE TABLE Company (cid INTEGER PRIMARY KEY, open_roles INTEGER))	500	100	• Index 4: Unclustered(cid) • Index 5: Clustered(open roles). There are 500 unique values in the range [1, 500]

Question 1a

Selectivity 1:

1/max(25000, 25000) = 1/25000. There are exactly 25000 values in Student.sid, and due to the foreign key, there are at most 25000 values of Application.sid.

```
SELECT Student.name, Company.open_roles, Application.referral
FROM Student, Application, Company
WHERE Student.sid = Application.sid                    -- (Selectivity 1)
AND Application.cid = Company.cid                      -- (Selectivity 2)
AND Student.semesters_completed > 6                    -- (Selectivity 3)
AND (Student.major='EECS' OR Company.open_roles <= 50) -- (Selectivity 4)
AND NOT Application.status = 'limbo'                   -- (Selectivity 5)
ORDER BY Company.open_roles;
```

Question 1b

Selectivity 2:

1/max(500, 500) = 1/500. Similarly to Selectivity 1, there are exactly 500 values in Company.cid, and due to the foreign key, there are at most 500 values in Application.cid.

```
SELECT Student.name, Company.open_roles, Application.referral
FROM Student, Application, Company
WHERE Student.sid = Application.sid                    -- (Selectivity 1)
AND Application.cid = Company.cid                      -- (Selectivity 2)
AND Student.semesters_completed > 6                    -- (Selectivity 3)
AND (Student.major='EECS' OR Company.open_roles <= 50) -- (Selectivity 4)
AND NOT Application.status = 'limbo'                   -- (Selectivity 5)
ORDER BY Company.open_roles;
```

Question 1c

Selectivity 3:

(10 - 6) / (10 - 0 + 1) = 4/11. We have 11 unique values, assumed to be equally distributed. Therefore we use the equation for less than or equal to which is (high key - value) / (high key - low key + 1).

```
SELECT Student.name, Company.open_roles, Application.referral
FROM Student, Application, Company
WHERE Student.sid = Application.sid                    -- (Selectivity 1)
AND Application.cid = Company.cid                      -- (Selectivity 2)
AND Student.semesters_completed > 6                    -- (Selectivity 3)
AND (Student.major='EECS' OR Company.open_roles <= 50) -- (Selectivity 4)
AND NOT Application.status = 'limbo'                   -- (Selectivity 5)
ORDER BY Company.open_roles;
```

Question 1d

Selectivity 4:

(1/130 + 1/10) - (1/130 * 1/10) = 10/1300 + 130/1300 - 1/1300 = 139/1300. We can find the selectivity that they are an EECS major by using the equation 1/distinct values. Next, we find the selectivity that open positions are less than or equal to 50 using the equation (v - low key) / ((high key - low key + 1) + (1 / number distinct)). Lastly we combine these two selectivities using S(p1) + S(p2) - S(p1)S(p2) to determine the selectivity of having one or the other

```
SELECT Student.name, Company.open_roles, Application.referral
FROM Student, Application, Company
WHERE Student.sid = Application.sid                    -- (Selectivity 1)
AND Application.cid = Company.cid                      -- (Selectivity 2)
AND Student.semesters_completed > 6                    -- (Selectivity 3)
AND (Student.major='EECS' OR Company.open_roles <= 50) -- (Selectivity 4)
AND NOT Application.status = 'limbo'                   -- (Selectivity 5)
ORDER BY Company.open_roles;
```

Question 1e

Selectivity 5:

1 - (1/10) = 9/10. Given 10 unique values, the non-negated predicate has selectivity 1/10, so we can use the equation for NOT which is 1 - selectivity of the predicate. The selectivity of the predicate is 1/10 (because there are 10 unique values).

```
SELECT Student.name, Company.open_roles, Application.referral
FROM Student, Application, Company
WHERE Student.sid = Application.sid                    -- (Selectivity 1)
AND Application.cid = Company.cid                      -- (Selectivity 2)
AND Student.semesters_completed > 6                    -- (Selectivity 3)
AND (Student.major='EECS' OR Company.open_roles <= 50) -- (Selectivity 4)
AND NOT Application.status = 'limbo'                   -- (Selectivity 5)
ORDER BY Company.open_roles;
```

Question 2

For each predicate, which is the first pass of Selinger's algorithm that uses its selectivity to estimate output size? (Pass 1, 2 or 3?)

Solution: Pass 2, Pass 2, Pass 1, Pass 3, Pass 1. C and E are pass 1 because they only involve filtering one table. A and B are pass 2 because they represent a join. Note that (d)—the OR predicate—is over 2 tables that have no associated join predicate, so the selection is postponed along with the cross-product, until after 3-way joins are done

```
SELECT Student.name, Company.open_roles, Application.referral
FROM Student, Application, Company
WHERE Student.sid = Application.sid                    -- (Selectivity 1)
AND Application.cid = Company.cid                      -- (Selectivity 2)
AND Student.semesters_completed > 6                    -- (Selectivity 3)
AND (Student.major='EECS' OR Company.open_roles <= 50) -- (Selectivity 4)
AND NOT Application.status = 'limbo'                   -- (Selectivity 5)
ORDER BY Company.open_roles;
```

Question 3

Mark the choices for all access plans that would be considered in pass 2 of the Selinger algorithm.

A, B, E, and F will be considered because they are not cross products. They are joined on a condition, so some rows can be filtered out, making our intermediate relations smaller.

(a) Student ⋈ Application (800 IOs)
(b) Application ⋈ Student (750 IOs)
(c) Student ⋈ Company (470 IOs)
(d) Company ⋈ Student (525 IOs)
(e) Application ⋈ Company (600 IOs)
(f) Company ⋈ Application (575 IOs)

```
SELECT Student.name, Company.open_roles, Application.referral
FROM Student, Application, Company
WHERE Student.sid = Application.sid                          -- (Selectivity 1)
AND Application.cid = Company.cid                            -- (Selectivity 2)
AND Student.semesters_completed > 6                          -- (Selectivity 3)
AND (Student.major='EECS' OR Company.open_roles <= 50)       -- (Selectivity 4)
AND NOT Application.status = 'limbo'                         -- (Selectivity 5)
ORDER BY Company.open_roles;
```

Question 4

Which choices from the previous question for all access plans would be chosen at the end of pass 2 of the Selinger algorithm?

B and F will be chosen because they have the lower cost for joining the two tables, and we have the assumption that our optimizer does not consider interesting orders. (Even if we did, there are no interesting orders in the other joins.)

(a) Student ⋈ Application (800 IOs)
(b) Application ⋈ Student (750 IOs)
(c) Student ⋈ Company (470 IOs)
(d) Company ⋈ Student (525 IOs)
(e) Application ⋈ Company (600 IOs)
(f) Company ⋈ Application (575 IOs)

```
SELECT Student.name, Company.open_roles, Application.referral
FROM Student, Application, Company
WHERE Student.sid = Application.sid                        -- (Selectivity 1)
AND Application.cid = Company.cid                          -- (Selectivity 2)
AND Student.semesters_completed > 6                        -- (Selectivity 3)
AND (Student.major='EECS' OR Company.open_roles <= 50)     -- (Selectivity 4)
AND NOT Application.status = 'limbo'                       -- (Selectivity 5)
ORDER BY Company.open_roles;
```

Question 5

Which plans that would be considered in pass 3?

F and H only. A-E can be immediately discarded because they aren't left-deep. G won't be considered because we chose (Company ⋈ Application) in pass 2. Similarly, choice I wouldn't be considered because we choose Application ⋈ Student in the previous pass. Choice J wouldn't be considered because there is no join condition on Student and Company, so this is a cross-join, which we avoid since we have other options.

(a) Company ⋈ (Application ⋈ Student) (175,000 IOs)
(b) Company ⋈ (Student ⋈ Application) (150,000 IOs)
(c) Application ⋈ (Company ⋈ Student) (155,000 IOs)
(d) Application ⋈ (Company ⋈ Student) (160,000 IOs)
(e) Student ⋈ (Company ⋈ Application) (215,000 IOs)
(f) (Company ⋈ Application) ⋈ Student (180,000 IOs)
(g) (Application ⋈ Company) ⋈ Student (200,000 IOs)
(h) (Application ⋈ Student) ⋈ Company (194,000 IOs)
(i) (Student ⋈ Application) ⋈ Company (195,000 IOs)
(j) (Student ⋈ Company) ⋈ Application (165,000 IOs)

```
SELECT Student.name, Company.open_roles, Application.referral
FROM Student, Application, Company
WHERE Student.sid = Application.sid                    -- (Selectivity 1)
AND Application.cid = Company.cid                      -- (Selectivity 2)
AND Student.semesters_completed > 6                    -- (Selectivity 3)
AND (Student.major='EECS' OR Company.open_roles <= 50) -- (Selectivity 4)
AND NOT Application.status = 'limbo'                   -- (Selectivity 5)
ORDER BY Company.open_roles;
```

Question 6

Which choice from the previous question for all plans would be chosen at the end of pass 3?

F, since F has the lower I/O cost between F and H.

(a) Company ⋈ (Application ⋈ Student) (175,000 IOs)
(b) Company ⋈ (Student ⋈ Application) (150,000 IOs)
(c) Application ⋈ (Company ⋈ Student) (155,000 IOs)
(d) Application ⋈ (Company ⋈ Student) (160,000 IOs)
(e) Student ⋈ (Company ⋈ Application) (215,000 IOs)
(f) (Company ⋈ Application) ⋈ Student (180,000 IOs)
(g) (Application ⋈ Company) ⋈ Student (200,000 IOs)
(h) (Application ⋈ Student) ⋈ Company (194,000 IOs)
(i) (Student ⋈ Application) ⋈ Company (195,000 IOs)
(j) (Student ⋈ Company) ⋈ Application (165,000 IOs)

```
SELECT Student.name, Company.open_roles, Application.referral
FROM Student, Application, Company
WHERE Student.sid = Application.sid                    -- (Selectivity 1)
AND Application.cid = Company.cid                      -- (Selectivity 2)
AND Student.semesters_completed > 6                    -- (Selectivity 3)
AND (Student.major='EECS' OR Company.open_roles <= 50) -- (Selectivity 4)
AND NOT Application.status = 'limbo'                   -- (Selectivity 5)
ORDER BY Company.open_roles;
```

Query Optimization 2

Query Optimization 2.1

True or False: When evaluating potential query plans, the set of left deep join plans are always guaranteed to contain the best plan

False: this is a heuristic that System R uses to shrink the search space

Query Optimization 2.1

True or False: As a heuristic, the System R optimizer avoids cross-products if possible

True

Query Optimization 2.1

True or False: A plan can result in an interesting order if it involves a sort-merge join

True: Sort-merge join leaves the joined tables in a sorted order which may be useful for future passes and/or if the query includes an ORDER BY clause

Query Optimization 2.1

True or False: The System R algorithm is greedy because for each pass, it only keeps the lowest cost plan for each combination of tables

False: it is not greedy because it keeps track of interesting orders which may not be the lowest cost plan, but may result in a lower cost plan in future passes (hence the use of dynamic programming)

Query Optimization 2.2

CREATE TABLE Flight (fid INTEGER PRIMARY KEY, from_id INTEGER REFERENCES City, to_id INTEGER REFERENCES City, aid INTEGER REFERENCES Airline)	NTuples: 100K, NPages: 50 Index: (I) unclustered B+-tree on aid. 20 leaf pages. (II) clustered B+-tree on (from_cid, fid). 10 leaf pages.
CREATE TABLE City (cid INTEGER PRIMARY KEY, name VARCHAR(16), state VARCHAR(16), population INTEGER)	NTuples: 50K, NPages: 20 Index: (III) clustered B+-tree on population. 10 leaf pages. (IV) unclustered index on cid. 5 leaf pages. Statistics: state in [1, 50], population in [10^6, $8*10^6$]
CREATE TABLE Airline (aid INTEGER PRIMARY KEY, hq_cid INTEGER REFERENCES City, name VARCHAR(16))	NTuples: 5K, NPages: 2

Assuming that (a) System R assumptions about uniformity and independence from lecture hold, and
(b) Primary key IDs are sequential, starting from 1

What is the selectivity of each predicate in the WHERE clause of the following query?

```
SELECT *
FROM Flight F, City C, Airline A
WHERE F.to_id = C.cid
AND F.aid = A.aid
AND F.aid >= 2500
AND C.population > 5e6
AND C.state = 'California';
```

Query Optimization 2.2

CREATE TABLE Flight (fid INTEGER PRIMARY KEY, from_id INTEGER REFERENCES City, to_id INTEGER REFERENCES City, aid INTEGER REFERENCES Airline)	NTuples: 100K, NPages: 50 Index: (I) unclustered B+-tree on aid. 20 leaf pages. (II) clustered B+-tree on (from_cid, fid). 10 leaf pages.
CREATE TABLE City (cid INTEGER PRIMARY KEY, name VARCHAR(16), state VARCHAR(16), population INTEGER)	NTuples: 50K, NPages: 20 Index: (III) clustered B+-tree on population. 10 leaf pages. (IV) unclustered index on cid. 5 leaf pages. Statistics: state in [1, 50], population in [10^6, $8*10^6$]
CREATE TABLE Airline (aid INTEGER PRIMARY KEY, hq_cid INTEGER REFERENCES City, name VARCHAR(16))	NTuples: 5K, NPages: 2

Assuming that (a) System R assumptions about uniformity and independence from lecture hold, and
(b) Primary key IDs are sequential, starting from 1

What is the selectivity of each predicate in the WHERE clause of the following query?

```
SELECT *
FROM Flight F, City C, Airline A
WHERE F.to_id = C.cid
AND F.aid = A.aid
AND F.aid >= 2500
AND C.population > 5e6
AND C.state = 'California';
```

`F.to_id = C.id`

50k tuples in City table, C.id is a PK and F.to_id is an FK, so there are 50k unique values of each. Apply formula (1 / MAX(nkeys(tab1), nkeys(tab2)))
= 1 / MAX(50k, 50k) = **1 / 50000**

Query Optimization 2.2

CREATE TABLE Flight (fid INTEGER PRIMARY KEY, from_id INTEGER REFERENCES City, to_id INTEGER REFERENCES City, aid INTEGER REFERENCES Airline)	NTuples: 100K, NPages: 50 Index: (I) unclustered B+-tree on aid. 20 leaf pages. (II) clustered B+-tree on (from_cid, fid). 10 leaf pages.
CREATE TABLE City (cid INTEGER PRIMARY KEY, name VARCHAR(16), state VARCHAR(16), population INTEGER)	NTuples: 50K, NPages: 20 Index: (III) clustered B+-tree on population. 10 leaf pages. (IV) unclustered index on cid. 5 leaf pages. Statistics: state in [1, 50], population in [10^6, $8*10^6$]
CREATE TABLE Airline (aid INTEGER PRIMARY KEY, hq_cid INTEGER REFERENCES City, name VARCHAR(16))	NTuples: 5K, NPages: 2

Assuming that (a) System R assumptions about uniformity and independence from lecture hold, and
(b) Primary key IDs are sequential, starting from 1

What is the selectivity of each predicate in the WHERE clause of the following query?

```
SELECT *
FROM Flight F, City C, Airline A
WHERE F.to_id = C.cid
AND F.aid = A.aid
AND F.aid >= 2500
AND C.population > 5e6
AND C.state = 'California';
```

`F.aid = A.aid`

5k tuples in Airline table, A.aid is PK, F.aid is FK so there are 5k values of each. Apply same formula as before:

= 1 / MAX(5k, 5k) = **1 / 5000**

Query Optimization 2.2

CREATE TABLE Flight (fid INTEGER PRIMARY KEY, from_id INTEGER REFERENCES City, to_id INTEGER REFERENCES City, aid INTEGER REFERENCES Airline)	NTuples: 100K, NPages: 50 Index: (I) unclustered B+-tree on aid. 20 leaf pages. (II) clustered B+-tree on (from_cid, fid). 10 leaf pages.
CREATE TABLE City (cid INTEGER PRIMARY KEY, name VARCHAR(16), state VARCHAR(16), population INTEGER)	NTuples: 50K, NPages: 20 Index: (III) clustered B+-tree on population. 10 leaf pages. (IV) unclustered index on cid. 5 leaf pages. Statistics: state in [1, 50], population in [10^6, $8*10^6$]
CREATE TABLE Airline (aid INTEGER PRIMARY KEY, hq_cid INTEGER REFERENCES City, name VARCHAR(16))	NTuples: 5K, NPages: 2

Assuming that (a) System R assumptions about uniformity and independence from lecture hold, and
(b) Primary key IDs are sequential, starting from 1

What is the selectivity of each predicate in the WHERE clause of the following query?

```
SELECT *
FROM Flight F, City C, Airline A
WHERE F.to_id = C.cid
AND F.aid = A.aid
AND F.aid >= 2500
AND C.population > 5e6
AND C.state = 'California';
```

F.aid >= 2500

5k values of F.aid (it is a PK for table of 5k tuples); values are 1-5000, so we can apply adjusted “>=” formula:
(High(col)-value)/(High(col)-Low(col) + 1) + 1/High(col)
= (5000 - 2500)/(5000-1+1) + 1/5000
= **2501/5000**

Query Optimization 2.2

CREATE TABLE Flight (fid INTEGER PRIMARY KEY, from_id INTEGER REFERENCES City, to_id INTEGER REFERENCES City, aid INTEGER REFERENCES Airline)	NTuples: 100K, NPages: 50 Index: (I) unclustered B+-tree on aid. 20 leaf pages. (II) clustered B+-tree on (from_cid, fid). 10 leaf pages.
CREATE TABLE City (cid INTEGER PRIMARY KEY, name VARCHAR(16), state VARCHAR(16), population INTEGER)	NTuples: 50K, NPages: 20 Index: (III) clustered B+-tree on population. 10 leaf pages. (IV) unclustered index on cid. 5 leaf pages. Statistics: state in [1, 50], population in [10^6, $8*10^6$]
CREATE TABLE Airline (aid INTEGER PRIMARY KEY, hq_cid INTEGER REFERENCES City, name VARCHAR(16))	NTuples: 5K, NPages: 2

Assuming that (a) System R assumptions about uniformity and independence from lecture hold, and
(b) Primary key IDs are sequential, starting from 1

What is the selectivity of each predicate in the WHERE clause of the following query?

```
SELECT *
FROM Flight F, City C, Airline A
WHERE F.to_id = C.cid
AND F.aid = A.aid
AND F.aid >= 2500
AND C.population > 5e6
AND C.state = 'California';
```

C.population > 5e6

Use range given in description of City table and apply formula for ">":

(High(col)-value)/(High(col)-Low(col) + 1) + 1/High(col)

= (8e6 - 5e6)/(8e6-1e6 + 1)

= 3e6/(7e6 + 1)

= 3000000/7000001

Query Optimization 2.2

CREATE TABLE Flight (fid INTEGER PRIMARY KEY, from_id INTEGER REFERENCES City, to_id INTEGER REFERENCES City, aid INTEGER REFERENCES Airline)	NTuples: 100K, NPages: 50 Index: (I) unclustered B+-tree on aid. 20 leaf pages. (II) clustered B+-tree on (from_cid, fid). 10 leaf pages.
CREATE TABLE City (cid INTEGER PRIMARY KEY, name VARCHAR(16), state VARCHAR(16), population INTEGER)	NTuples: 50K, NPages: 20 Index: (III) clustered B+-tree on population. 10 leaf pages. (IV) unclustered index on cid. 5 leaf pages. Statistics: state in [1, 50], population in [10^6, $8*10^6$]
CREATE TABLE Airline (aid INTEGER PRIMARY KEY, hq_cid INTEGER REFERENCES City, name VARCHAR(16))	NTuples: 5K, NPages: 2

Assuming that (a) System R assumptions about uniformity and independence from lecture hold, and
(b) Primary key IDs are sequential, starting from 1

What is the selectivity of each predicate in the WHERE clause of the following query?

```
SELECT *
FROM Flight F, City C, Airline A
WHERE F.to_id = C.cid
AND F.aid = A.aid
AND F.aid >= 2500
AND C.population > 5e6
AND C.state = 'California';
```

C.state = 'California'
There are 50 states, so
= 1/50

Query Optimization 2.3

`CREATE TABLE Flight (`
`fid INTEGER PRIMARY KEY,`
`from_id INTEGER REFERENCES City,`
`to_id INTEGER REFERENCES City,`
`aid INTEGER REFERENCES Airline)`
	NTuples: 100K, NPages: 50
	Index:
	(I) unclustered B+-tree on aid. 20 leaf pages.
	(II) clustered B+-tree on (from_cid, fid). 10 leaf pages.

`CREATE TABLE City (`
`cid INTEGER PRIMARY KEY,`
`name VARCHAR(16),`
`state VARCHAR(16),`
`population INTEGER)`
	NTuples: 50K, NPages: 20
	Index:
	(III) clustered B+-tree on population. 10 leaf pages. (IV) unclustered index on cid. 5 leaf pages. Statistics:
	state in [1, 50], population in [10^6, $8*10^6$]

`CREATE TABLE Airline (`
`aid INTEGER PRIMARY KEY,`
`hq_cid INTEGER REFERENCES City,`
`name VARCHAR(16))`
	NTuples: 5K, NPages: 2

Assuming that (a) System R assumptions about uniformity and independence from lecture hold, and
(b) Primary key IDs are sequential, starting from 1
(c) Optimizer hasn't discarded rows
(d) B+ trees are of height 2

Fill in each blank in the dynamic programming table for a System R Pass 1

```
SELECT *
FROM Flight F, City C, Airline A
WHERE F.to_id = C.cid
AND F.aid = A.aid
AND F.aid >= 2500
AND C.population > 5e6
AND C.state = 'California';
```

Table(s)	Plans	Interesting Orders from Plan (N/A if none)	Cost (I/Os)
Flight	Index (I)		
City	Filescan		
City	Index (III)		

Query Optimization 2.3

`CREATE TABLE Flight (`
`fid INTEGER PRIMARY KEY,`
`from_id INTEGER REFERENCES City,`
`to_id INTEGER REFERENCES City,`
`aid INTEGER REFERENCES Airline)`
	NTuples: 100K, NPages: 50
	Index:
	(I) unclustered B+-tree on aid. 20 leaf pages.
	(II) clustered B+-tree on (from_cid, fid). 10 leaf pages.

`CREATE TABLE City (`
`cid INTEGER PRIMARY KEY,`
`name VARCHAR(16),`
`state VARCHAR(16),`
`population INTEGER)`
	NTuples: 50K, NPages: 20
	Index:
	(III) clustered B+-tree on population. 10 leaf pages. (IV) unclustered index on cid. 5 leaf pages. Statistics:
	state in [1, 50], population in [10^6, $8*10^6$]

`CREATE TABLE Airline (`
`aid INTEGER PRIMARY KEY,`
`hq_cid INTEGER REFERENCES City,`
`name VARCHAR(16))`
	NTuples: 5K, NPages: 2

Assuming that (a) System R assumptions about uniformity and independence from lecture hold, and

2 I/Os (read root, inner page of index)
+ (20 + 100k) (read 20 pg of index, 100k reads for unclustered)
* <selectivity factor for F.aid >= 2500 (R3)

Fill in e

```
SELECT *
FROM Flight F, City C, Airline A
WHERE F.to_id = C.cid
AND F.aid = A.aid
AND F.aid >= 2500
AND C.population > 5e6
AND C.state = 'California';
```

Table(s)	Plans	Interesting Orders from Plan (N/A if none)	Cost (I/Os)
Flight	Index (I)	aid	50032
City	Filescan		
City	Index (III)		

Query Optimization 2.3

`CREATE TABLE Flight (`
`fid INTEGER PRIMARY KEY,`
`from_id INTEGER REFERENCES City,`
`to_id INTEGER REFERENCES City,`
`aid INTEGER REFERENCES Airline)`
	NTuples: 100K, NPages: 50
	Index:
	(I) unclustered B+-tree on aid. 20 leaf pages.
	(II) clustered B+-tree on (from_cid, fid). 10 leaf pages.

`CREATE TABLE City (`
`cid INTEGER PRIMARY KEY,`
`name VARCHAR(16),`
`state VARCHAR(16),`
`population INTEGER)`
	NTuples: 50K, NPages: 20
	Index:
	(III) clustered B+-tree on population. 10 leaf pages. (IV) unclustered index on cid. 5 leaf pages. Statistics:
	state in [1, 50], population in [10^6, $8*10^6$]

`CREATE TABLE Airline (`
`aid INTEGER PRIMARY KEY,`
`hq_cid INTEGER REFERENCES City,`
`name VARCHAR(16))`
	NTuples: 5K, NPages: 2

Assuming that (a) System R assumptions about uniformity and independence from lecture hold, and

20 I/Os (no index used, just read the pages)

Fill in e

```
SELECT *
FROM Flight F, City C, Airline A
WHERE F.to_id = C.cid
AND F.aid = A.aid
AND F.aid >= 2500
AND C.population > 5e6
AND C.state = 'California';
```

Table(s)	Plans	Interesting Orders from Plan (N/A if none)	Cost (I/Os)
Flight	Index (I)	aid	50032
City	Filescan	N/A	20
City	Index (III)		

Query Optimization 2.3

`CREATE TABLE Flight (`
`fid INTEGER PRIMARY KEY,`
`from_id INTEGER REFERENCES City,`
`to_id INTEGER REFERENCES City,`
`aid INTEGER REFERENCES Airline)`
	NTuples: 100K, NPages: 50
	Index:
	(I) unclustered B+-tree on aid. 20 leaf pages.
	(II) clustered B+-tree on (from_cid, fid). 10 leaf pages.

`CREATE TABLE City (`
`cid INTEGER PRIMARY KEY,`
`name VARCHAR(16),`
`state VARCHAR(16),`
`population INTEGER)`
	NTuples: 50K, NPages: 20
	Index:
	(III) clustered B+-tree on population. 10 leaf pages. (IV) unclustered index on cid. 5 leaf pages. Statistics:
	state in [1, 50], population in [10^6, $8*10^6$]

`CREATE TABLE Airline (`
`aid INTEGER PRIMARY KEY,`
`hq_cid INTEGER REFERENCES City,`
`name VARCHAR(16))`
	NTuples: 5K, NPages: 2

Assuming that (a) System R assumptions about uniformity and independence from lecture hold, and

2 I/Os (read root, inner node of index)
+ (10 + 20) (read 10 pages of index, 20 pages of data)
* <selectivity factor for C.population > 5e6>

Fill in e

```
SELECT *
FROM Flight F, City C, Airline A
WHERE F.to_id = C.cid
AND F.aid = A.aid
AND F.aid >= 2500
AND C.population > 5e6
AND C.state = 'California';
```

Table(s)	Plans	Interesting Orders from Plan (N/A if none)	Cost (I/Os)
Flight	Index (I)	aid	50032
City	Filescan	N/A	20
City	Index (III)	N/A	15

Query Optimization 2.4

`CREATE TABLE Flight (`
`fid INTEGER PRIMARY KEY,`
`from_id INTEGER REFERENCES City,`
`to_id INTEGER REFERENCES City,`
`aid INTEGER REFERENCES Airline)`
	NTuples: 100K, NPages: 50
	Index:
	(I) unclustered B+-tree on aid. 20 leaf pages.
	(II) clustered B+-tree on (from_cid, fid). 10 leaf pages.

`CREATE TABLE City (`
`cid INTEGER PRIMARY KEY,`
`name VARCHAR(16),`
`state VARCHAR(16),`
`population INTEGER)`
	NTuples: 50K, NPages: 20
	Index:
	(III) clustered B+-tree on population. 10 leaf pages. (IV) unclustered index on cid. 5 leaf pages. Statistics:
	state in [1, 50], population in [10^6, $8*10^6$]

`CREATE TABLE Airline (`
`aid INTEGER PRIMARY KEY,`
`hq_cid INTEGER REFERENCES City,`
`name VARCHAR(16))`
	NTuples: 5K, NPages: 2

After pass 2, which of the following plans could be in the dynamic programming table?

1) City [Index(III)] JOIN Airline [File scan]
2) City [Index(III)] JOIN Flight [Index(I)]
3) Flight [Index(II)] JOIN City [Index(III)]

```
SELECT *
FROM Flight F, City C, Airline A
WHERE F.to_id = C.cid
AND F.aid = A.aid
AND F.aid >= 2500
AND C.population > 5e6
AND C.state = 'California';
```

Table(s)	Plans	Interesting Orders from Plan (N/A if none)	Cost (I/Os)
Flight	Index (I)	aid	50032
City	Filescan	N/A	20
City	Index (III)	N/A	15

Query Optimization 2.4

`CREATE TABLE Flight (`
`fid INTEGER PRIMARY KEY,`
`from_id INTEGER REFERENCES City,`
`to_id INTEGER REFERENCES City,`
`aid INTEGER REFERENCES Airline)`
	NTuples: 100K, NPages: 50
	Index:
	(I) unclustered B+-tree on aid. 20 leaf pages.
	(II) clustered B+-tree on (from_cid, fid). 10 leaf pages.

`CREATE TABLE City (`
`cid INTEGER PRIMARY KEY,`
`name VARCHAR(16),`
`state VARCHAR(16),`
`population INTEGER)`
	NTuples: 50K, NPages: 20
	Index:
	(III) clustered B+-tree on population. 10 leaf pages. (IV) unclustered index on cid. 5 leaf pages. Statistics:
	state in [1, 50], population in [10^6, $8*10^6$]

`CREATE TABLE Airline (`
`aid INTEGER PRIMARY KEY,`
`hq_cid INTEGER REFERENCES City,`
`name VARCHAR(16))`
	NTuples: 5K, NPages: 2

After pass 2, which of the following plans could be in the dynamic programming table?

1) City [Index(III)] JOIN Airline [File scan] **No**: no condition joins City, Airline so this would be a cross product
2) City [Index(III)] JOIN Flight [Index(I)]
3) Flight [Index(II)] JOIN City [Index(III)]

```
SELECT *
FROM Flight F, City C, Airline A
WHERE F.to_id = C.cid
AND F.aid = A.aid
AND F.aid >= 2500
AND C.population > 5e6
AND C.state = 'California';
```

Table(s)	Plans	Interesting Orders from Plan (N/A if none)	Cost (I/Os)
Flight	Index (I)	aid	50032
City	Filescan	N/A	20
City	Index (III)	N/A	15

Query Optimization 2.4

`CREATE TABLE Flight (`
`fid INTEGER PRIMARY KEY,`
`from_id INTEGER REFERENCES City,`
`to_id INTEGER REFERENCES City,`
`aid INTEGER REFERENCES Airline)`
	NTuples: 100K, NPages: 50
	Index:
	(I) unclustered B+-tree on aid. 20 leaf pages.
	(II) clustered B+-tree on (from_cid, fid). 10 leaf pages.

`CREATE TABLE City (`
`cid INTEGER PRIMARY KEY,`
`name VARCHAR(16),`
`state VARCHAR(16),`
`population INTEGER)`
	NTuples: 50K, NPages: 20
	Index:
	(III) clustered B+-tree on population. 10 leaf pages. (IV) unclustered index on cid. 5 leaf pages. Statistics:
	state in [1, 50], population in [10^6, $8*10^6$]

`CREATE TABLE Airline (`
`aid INTEGER PRIMARY KEY,`
`hq_cid INTEGER REFERENCES City,`
`name VARCHAR(16))`
	NTuples: 5K, NPages: 2

After pass 2, which of the following plans could be in the dynamic programming table?

1) City [Index(III)] JOIN Airline [File scan] **No**: no condition joins City, Airline so this would be a cross product
2) City [Index(III)] JOIN Flight [Index(I)] **Yes**: City[Index(III)] kept for lowest cost, Flight [Index(I)] kept b.c. interesting order
3) Flight [Index(II)] JOIN City [Index(III)]

```
SELECT *
FROM Flight F, City C, Airline A
WHERE F.to_id = C.cid
AND F.aid = A.aid
AND F.aid >= 2500
AND C.population > 5e6
AND C.state = 'California';
```

Table(s)	Plans	Interesting Orders from Plan (N/A if none)	Cost (I/Os)
Flight	Index (I)	aid	50032
City	Filescan	N/A	20
City	Index (III)	N/A	15

Query Optimization 2.4

`CREATE TABLE Flight (`
`fid INTEGER PRIMARY KEY,`
`from_id INTEGER REFERENCES City,`
`to_id INTEGER REFERENCES City,`
`aid INTEGER REFERENCES Airline)`
	NTuples: 100K, NPages: 50
	Index:
	(I) unclustered B+-tree on aid. 20 leaf pages.
	(II) clustered B+-tree on (from_cid, fid). 10 leaf pages.

`CREATE TABLE City (`
`cid INTEGER PRIMARY KEY,`
`name VARCHAR(16),`
`state VARCHAR(16),`
`population INTEGER)`
	NTuples: 50K, NPages: 20
	Index:
	(III) clustered B+-tree on population. 10 leaf pages. (IV) unclustered index on cid. 5 leaf pages. Statistics:
	state in [1, 50], population in [10^6, $8*10^6$]

`CREATE TABLE Airline (`
`aid INTEGER PRIMARY KEY,`
`hq_cid INTEGER REFERENCES City,`
`name VARCHAR(16))`
	NTuples: 5K, NPages: 2

After pass 2, which of the following plans could be in the dynamic programming table?

1) City [Index(III)] JOIN Airline [File scan] **No**: no condition joins City, Airline so this would be a cross product
2) City [Index(III)] JOIN Flight [Index(I)] **Yes**: City[Index(III)] kept for lowest cost, Flight [Index(I)] kept b.c. interesting order
3) Flight [Index(II)] JOIN City [Index(III)] **No:** Flight[Index(II)] would not be kept from Pass 1; it is more expensive than full scan and has no interesting orders

```
SELECT *
FROM Flight F, City C, Airline A
WHERE F.to_id = C.cid
AND F.aid = A.aid
AND F.aid >= 2500
AND C.population > 5e6
AND C.state = 'California';
```

Table(s)	Plans	Interesting Orders from Plan (N/A if none)	Cost (I/Os)
Flight	Index (I)	aid	50032
City	Filescan	N/A	20
City	Index (III)	N/A	15

Query Optimization 2.5

`CREATE TABLE Flight (`
`fid INTEGER PRIMARY KEY,`
`from_id INTEGER REFERENCES City,`
`to_id INTEGER REFERENCES City,`
`aid INTEGER REFERENCES Airline)`
	NTuples: 100K, NPages: 50
	Index:
	(I) unclustered B+-tree on aid. 20 leaf pages.
	(II) clustered B+-tree on (from_cid, fid). 10 leaf pages.

`CREATE TABLE City (`
`cid INTEGER PRIMARY KEY,`
`name VARCHAR(16),`
`state VARCHAR(16),`
`population INTEGER)`
	NTuples: 50K, NPages: 20
	Index:
	(III) clustered B+-tree on population. 10 leaf pages. (IV) unclustered index on cid. 5 leaf pages. Statistics:
	state in [1, 50], population in [10^6, $8*10^6$]

`CREATE TABLE Airline (`
`aid INTEGER PRIMARY KEY,`
`hq_cid INTEGER REFERENCES City,`
`name VARCHAR(16))`
	NTuples: 5K, NPages: 2

Suppose we want to optimize for queries similar to the query below; which of the following suggestions could reduce I/O cost?

1) Change Index(III) to be unclustered

2) Store City as a sorted file on population

```
SELECT *
FROM Flight F, City C, Airline A
WHERE F.to_id = C.cid
AND F.aid = A.aid
AND F.aid >= 2500
AND C.population > 5e6
AND C.state = 'California';
```

Query Optimization 2.5

`CREATE TABLE Flight (`
`fid INTEGER PRIMARY KEY,`
`from_id INTEGER REFERENCES City,`
`to_id INTEGER REFERENCES City,`
`aid INTEGER REFERENCES Airline)`
	NTuples: 100K, NPages: 50
	Index:
	(I) unclustered B+-tree on aid. 20 leaf pages.
	(II) clustered B+-tree on (from_cid, fid). 10 leaf pages.

`CREATE TABLE City (`
`cid INTEGER PRIMARY KEY,`
`name VARCHAR(16),`
`state VARCHAR(16),`
`population INTEGER)`
	NTuples: 50K, NPages: 20
	Index:
	(III) clustered B+-tree on population. 10 leaf pages. (IV) unclustered index on cid. 5 leaf pages. Statistics:
	state in [1, 50], population in [10^6, $8*10^6$]

`CREATE TABLE Airline (`
`aid INTEGER PRIMARY KEY,`
`hq_cid INTEGER REFERENCES City,`
`name VARCHAR(16))`
	NTuples: 5K, NPages: 2

Suppose we want to optimize for queries similar to the query below; which of the following suggestions could reduce I/O cost?

1) Change Index(III) to be unclustered: Won't reduce I/O cost. Unclustered index results in more random accesses so we might load a page more than once
2) Store City as a sorted file on population

```
SELECT *
FROM Flight F, City C, Airline A
WHERE F.to_id = C.cid
AND F.aid = A.aid
AND F.aid >= 2500
AND C.population > 5e6
AND C.state = 'California';
```

Query Optimization 2.5

`CREATE TABLE Flight (`
`fid INTEGER PRIMARY KEY,`
`from_id INTEGER REFERENCES City,`
`to_id INTEGER REFERENCES City,`
`aid INTEGER REFERENCES Airline)`
	NTuples: 100K, NPages: 50
	Index:
	(I) unclustered B+-tree on aid. 20 leaf pages.
	(II) clustered B+-tree on (from_cid, fid). 10 leaf pages.

`CREATE TABLE City (`
`cid INTEGER PRIMARY KEY,`
`name VARCHAR(16),`
`state VARCHAR(16),`
`population INTEGER)`
	NTuples: 50K, NPages: 20
	Index:
	(III) clustered B+-tree on population. 10 leaf pages. (IV) unclustered index on cid. 5 leaf pages. Statistics:
	state in [1, 50], population in [10^6, $8*10^6$]

`CREATE TABLE Airline (`
`aid INTEGER PRIMARY KEY,`
`hq_cid INTEGER REFERENCES City,`
`name VARCHAR(16))`
	NTuples: 5K, NPages: 2

Suppose we want to optimize for queries similar to the query below; which of the following suggestions could reduce I/O cost?

1) Change Index(III) to be unclustered: Won't reduce I/O cost. Unclustered index results in more random accesses so we might load a page more than once
2) Store City as a sorted file on population: Could reduce I/O cost by making range lookups (e.g. population > 5e6) more efficient

```
SELECT *
FROM Flight F, City C, Airline A
WHERE F.to_id = C.cid
AND F.aid = A.aid
AND F.aid >= 2500
AND C.population > 5e6
AND C.state = 'California';
```

Please fill out our feedback form at https://tinyurl.com/CS186ExamPrep3FA20