Midterm Answers
Question 3: Query Optimization (32 points)

Schema:

```
Guitars (gid, brand, price)
Players (pid, name, age)
LastPlayed (gid, pid, date)
```

Query:

```
SELECT P.name
FROM Guitars G, Players P, LastPlayed L
WHERE G.gid = L.gid AND P.pid = L.pid
      AND P.age < 25 AND G.brand = 'Gibson'
      AND G.price > 3000;
```

Data distribution:

Players.age ranges from 10 to 85
Guitars.brand has 10 distinct values
Guitars.price ranges from 1,000 to 5,000
Guitars.gid has 1,000 distinct values
Players.pid has 1,000 distinct values

a. *(5 points)* Compute the selectivity for each individual term in the WHERE clause.

G.gid = L.gid : **1/1000**

RF = 1 / MAX(NKeys(G.gid), NKeys(L.gid))

Known:

- NKeys(G.gid) = 1000
- NKeys(L.gid) <= 1000 b/c foreign key referencing G.gid

RF = 1 / MAX(1000, X) where X <= 1000
RF = **1 / 1000**

P.pid = L.pid : **1/1000**

RF = 1 / MAX(NKeys(P.pid), NKeys(L.pid))

Known:

- NKeys(P.pid) = 1000
- NKeys(L.pid) <= 1000 b/c foreign key referencing P.pid

RF = 1 / MAX(1000, X) where X <= 1000
RF = **1 / 1000**

P.age < 25 : **1/5**

RF = (25 – Low(P.age)) / (High(P.age) – Low(P.age))

Known:
- Low(P.age) = 10
- High(P.age) = 85

RF = (25 – 10) / (85 – 10)
RF = **1 / 5**

G.brand = 'Gibson' : **1/10**

RF = 1 / NKeys(G.brand)

Known:
- NKeys(G.brand) = 10

RF = **1 / 10**

G.price > 3000 : **1/2**

RF = (High(G.price) – 3000) / (High(G.price) – Low(G.price))

Known:
- High(G.price) = 5000
- Low(G.price) = 1000

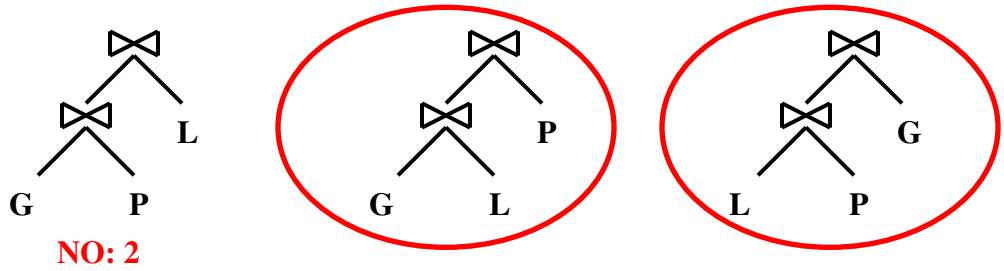
RF = (5000 – 3000) / (5000 – 1000)
RF = **1 / 2**

POINT BREAKDOWN: 1 point per correct reduction factor

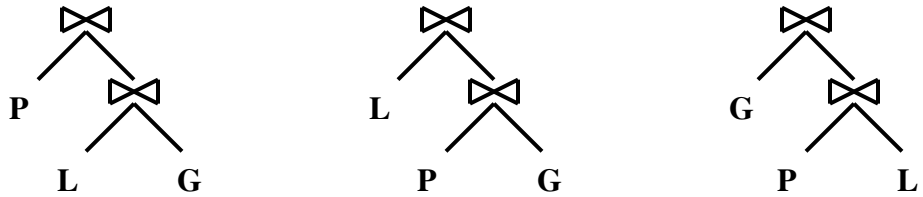
b. *(5 points)* Circle all join orders that would be considered by the System R optimizer.

The System R optimizer follows two primary heuristics:
1. It only considers left-deep plans
2. It avoids cross-products whenever possible

So, any right-deep plans will be ignored, as well as plans that directly join G and P (because it will result in a cross-product). The orders that are considered are circled, and the others are labeled with the above rule(s) that caused them to be discarded:



NO: 2



NO: 1 NO: 1, 2 NO: 1

POINT BREAKDOWN: 5 points for all 6 trees correctly marked
3 points for 5 trees correctly marked
1 point for 4 trees correctly marked
0 points for < 4 trees correctly marked

---

c. *(12 points)* Fill in the IO/tuple counts in the query plan tree.

The following information is given:

- B+ tree on Guitars.gid: unclustered, uses alternative 2 for data entries
- Average of 3 IO's to retrieve data entry from the index
- Guitars: 40 bytes/tuple, 100 tuples/page, 10 pages
- Players: 80 bytes/tuple, 50 tuples/page, 20 pages
- LastPlayed: 20 bytes/tuple, 200 tuples/page, 100 pages

Access Methods:

- Players: file scan
- LastPlayed: file scan
- Guitars: B+ tree on gid

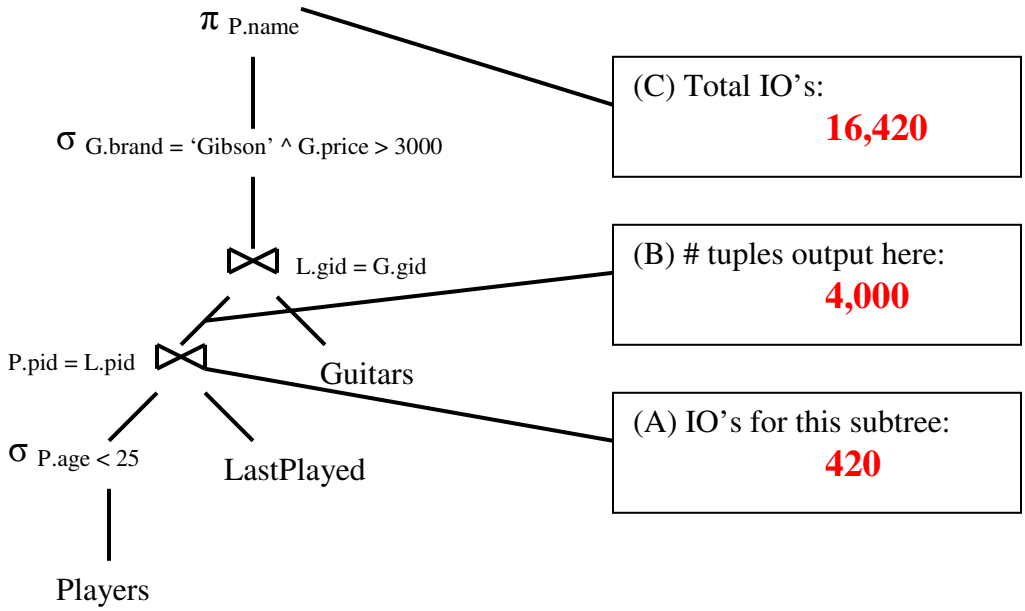
Join Algorithms:

- Players with LastPlayed: page-oriented nested loops join
- Subtree with Guitars: index nested loops join

Operator Implementations

- Selection (both instances): on the fly
- Projection: on the fly

The Plan:



A) Number of IOs:

= (Scan Players) + (Pages from Players in Join) * (Scan LastPlayed)
= (Pages in Players) + (Pages in Players after selection) * (Pages in LastPlayed)
= 20 + (1/5 * 20) * 100
= **420**

Note: You have to read each tuple off disk before you make the selection on P.age, so the scan of Players looks at all 20 pages. However, the join algorithm will scan LastPlayed once for every page of Players *that reaches the join*, so only 1/5 of the 20 pages.

B) Number of tuples:

= Total tuples times product of RF's
= (Tuples in Players) * (Tuples in LastPlayed) * (RF for select) * (RF for join)
= (50 * 20) * (200 * 100) * (1/5) * (1/1000)
= **4,000**

Note: Remember, since we care about the number of tuples that is output from the join, we also need to consider the reduction brought about by the join condition itself.

C) Number of IOs:

= (Scan subtree) + (Tuples in subtree) * (Cost to lookup matches in Guitars)
= (Pages in subtree) + (Tuples in subtree) * (Lookup data entry + get data record)
= 420 + 4000 * (3 + 1)
= **16,420**

Note: The cost to lookup a matching tuple in Guitars is the cost to retrieve the data entry (given as 3 IOs) and the cost to retrieve the actual data record from disk. Because G.gid is the primary key, you know that at most one tuples will have the correct matching value, so (clustered or unclustered) it will cost one IO per retrieval.

POINT BREAKDOWN: 4 points per correctly filled-in blank

d. *(10 points)* Say whether or not each change could have reduced the number of IOs.

Pushing down the selection on G.brand and G.price below the join: **NO**

> Since you're using index nested loops join, you're using the index to retrieve the matching tuples – filtering out after they've been taken from disk won't reduce the IO cost of that retrieval.

Creating a temporary file to store the results of the selection on P.age: **NO**

> You'll still have to scan all of Players at least once, and a temporary file won't reduce the number of pages that reach the join algorithm, so the cost function won't be affected.

Having the index on Guitars.gid be clustered: **NO**

> Because gid is the primary key of Guitars, you're guaranteed that at most one tuple will match the search key. So, clustering the index won't help because you'll never be reading more than one page's worth of tuples off disk at a time.

Projecting out P.age before the join: **YES**

> Though it won't affect the cost of scanning Players, it will reduce the number of pages that reach the join algorithm (because you'll be able to fit more tuples on a page), thus lowering the number of times you scan LastPlayed.

Changing the first join to block-oriented nested loops join: **YES**

> If you have the memory, block-oriented nested loops is cheaper than page-oriented nested loops join because you'll only have to scan LastPlayed for every *block* of Players, instead of every page. As long as you have at least 4 buffer frames, it will speed things up. And, since the question didn't ask if it definitely *would* (because that depends on the buffer pool size), but if it *could* help, the answer is yes.

POINT BREAKDOWN: 2 points per correctly filled-in blank