CS 186 - Fall 2024
Exam Prep Section 6 (Sol)
Iterators and Joins

Joins 1

We will be joining two tables: a table of students, and a table of assignment submissions; and we will be joining by the student ID:

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

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

For the questions below, consider the following query:

```
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**

Questions:

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

   Answer: 160,020 I/Os.
   The formula for a simple nested loop join is $[S] + |S| \cdot [A]$.
   Plugging in the numbers gives us $20 + (20 \cdot 200) \cdot 40 = 160,020$ I/Os.

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

   Answer: 200,040 I/Os.
   The formula for a simple nested loop join is $[A] + |A| \cdot [S]$.
   Plugging in the numbers gives us $40 + (40 \cdot 250) \cdot 20 = 200,040$ I/Os.

3. What is the I/O cost of a block nested loop join for Students on 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 $20 + 2 \cdot 40 = 100$ I/Os.

4. What about block nested loop join for AssignmentSubmissions on 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 $40 + 4 \cdot 20 = 120$ I/Os.

5. What is the I/O cost of an Index-Nested Loop Join for Students on AssignmentSubmissions?

   Assume we have a clustered alternative 2 index on AssignmentSubmissions.student id, in the form of a height 2 B+ tree. Assume that index node and leaf pages are not cached; all hits are on the same leaf page; and all hits are also on the same data page.

   Answer: 16,020 I/Os.
   The formula is $[S] + |S| \cdot$ (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 $20 + 4000 \cdot 4 = 16,020$ I/Os.

6. Now assume we have a unclustered alternative 2 index on AssignmentSubmissions.student id, 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 on AssignmentSubmissions?

   HINT: The foreign key in AssignmentSubmissions may play a role in how many accesses we do per record.

   Answer: 22,020 I/Os.
   The formula is $[S] + |S| \cdot$ *h*cost of index lookup*i*.
   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 in AssignmentSubmission exactly once (since each AssignmentSubmission is foreign-keyed on a student id) - so there will be $|A| = 10,000$ data page lookups, one for each row. So the total cost is $20 + 4000 \cdot 3 + 10000 = 22,020$ I/Os.

7. What is the cost of an unoptimized sort-merge join for Students on AssignmentSubmissions?

   Assume we have $B = 12$ buffer pages.

   Answer: 300 I/Os.
   The formula is *h*cost of sorting S*i* + *h*cost of sorting A*i* + $[S] + [A]$.
   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 $(2 \cdot 2[S]) + (2 \cdot 2[A]) + [S] + [A] = 5([S] + [A]) = 5 \cdot 60 = 300$ I/Os.

8. What is the cost of an optimized sort-merge join for Students on 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.

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* = 11? We would still have 2 + 4 < 11 − 1 runs.
   What about *B* = 10? We would still have 2 + 4 < 10 − 1 runs.
   What about *B* = 9? Now we have 3 runs for *S* and 5 runs for *A*, 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.

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

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

Questions:

1. What is the I/O cost of a simple nested loop join for joining Teams on 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.

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 page nested loop join is $[T] + [T] \cdot [P]$.
   Plugging in the numbers gives us $30 + 30 \cdot 300 = 9,030$ I/Os.

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 $[T] + d[T]/(B - 2)e \cdot [P]$.
   Plugging in the numbers gives us $30 + 4 \cdot 300 = 1,230$ I/Os.

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. The formula for an index nested loop join is $[P] + |P| \cdot$ cost to find matching records. For each record in Players, we will search our index for the corresponding record in the Teams table. 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 $300 + (300 \cdot 50) \cdot 3 = 45,300$ I/Os.

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 on Teams.team id = Players.team id using one of these indexes?

   Answer: 6,030 I/Os.
   Even though the index on Players.player id has a lower height, it is not useful for this join since the Players.player id column is not part of the join. We must use the clustered index of height 3 on Players.team id.
   If we are using an index on Players.team id, this means that Players should be the inner relation. The formula for an index nested loop join is $[T] + |T| \cdot$ cost to find matching records. For each record in Teams, we will search our index for the corresponding records in the Players table. 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 $30 + (30 \cdot 40) \cdot 5 = 6,030$ I/Os.

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 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 $30 + (30 \cdot 40) \cdot 29 = 34,830$ I/Os.

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? *Hint: Consider which of the joins are affected by duplicates. Remember that only one of the tables has duplicates, while the other does not.*

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.

For sort merge join, many duplicate values generally tend to increase the number of I/Os, since itera tors 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.

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 it should partition into similarly sized chunks, so there should not be significant differences in the number of I/Os compared to the original scenario.

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),
    ...
);
```

For the questions below, consider the following query:

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

We also have:

- Students has **[*S*] = 600 pages**, with p_S **= 60 records per page**
- AssignmentSubmissions has **[*A*] = 600 pages**, with p_A **= 60 records per page**
- **B = 10**

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 con straints)? Why?
   Anything that's not GHJ or Hash Join. This is because we have only 1 hash function and hence can not recursively partition. Remember for recursive partitioning, a new, independent hash function must be used in order to effectively re-partition the data.

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 so to create partitions that are $B2$ 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 signifi cant number of passes to sort a relation. This in turn will result in an increase in the overall IO cost.

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/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.

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 stu dent 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.