Assignment #3

CS 348 - Winter 2025

Due : 11:59 p.m., Sun, March 30, 2025

Appeal Deadline: One week after being returned

Submission Instruction

This assignment consists of 2 parts. Part I will be submitted through Crowdmark. See the website for more detailed instructions. In particular, do not forget to submit one file per question to make the lives of TAs easier. Part 2 is a programming question and requires submission of files through Marmoset.
For Part 1: You do not have to type the questions if you are using Latex. Just specifying the question number is good enough. You do not need to keep the same font style. Here is a draft/empty latex template you may download and use: `https://www.overleaf.com/read/pxmgghkqksvg#39ea01`. Handwritten solutions and scanned to pdf are also acceptable.

Part I

This part consists of 4 questions covering topics on query processing, indexes, optimization, and transactions.

Question 1.

[15 marks in total] Consider a table occupying 1,000,000 disk blocks/pages (recall we use the term block and page interchangeably). There is no index, and 101 memory blocks are available for query processing. Suppose you have the following two options to improve query performance:

(1) Buy more memory to increase the number of available memory blocks to 1001.

(2) Buy a faster disk to increase the speed of I/O by 10%.

(a) (8 marks) If the objective is to speed up the query "SELECT * FROM R WHERE R.A > 78;", which option is more effective? Briefly justify your answer.

(b) (7 marks) If the objective is to speed up the query "SELECT * FROM R ORDER BY A;", which option is more effective now? Briefly justify your answer.

Question 2.

[24 marks in total] Consider the following schema for an online bookstore:

```
Cust (custID, name, address, state, zip)
Book (bookID, title, author, price, category)
Order (orderID, custID, bookID, shipDate)
Inventory (bookID, quantity, warehouseID, shelfLocation)
Warehouse (warehouseID, state)
```

`Cust` and `Book` represent customers and books, respectively. When a customer buys a book, a tuple is entered into `Order`. `Inventory` records the quantity and shelf location of each book for every warehouse. `Warehouse` records the state where each warehouse is located in. `price` is numeric. `shipDate` is an integer representation of a date. In the following questions, `:today` is a constant denoting the integer representation of today's date.

(a) (8 points) Transform the following query into an equivalent query that 1) contains no cross products, and 2) performs projections and selections as early as possible. Represent your result as a relational algebra expression tree.

$$\pi_{\texttt{title,author}}\Big(\sigma_{\texttt{(state="NC")and(authorLIKE"\%Kondo")and(shipDate>:today-60)}}\Big(\sigma_{\texttt{(Cust.custID=Order.custID)and(Book.bookID=Order.bookID)}}\Big(\texttt{(Cust} \times \texttt{Order} \times \texttt{Book)})\Big)\Big)$$

For the following two parts of this question, we assume the following (partial) statistics:

- $|\texttt{Order}| = 60,000$; $|\pi_{\texttt{bookID}}\texttt{Order}| = 2,000$; $|\pi_{\texttt{shipDate}}\texttt{Order}| = 1,000$;
- $|\texttt{Inventory}| = 40,000$; $|\pi_{\texttt{bookID}}\texttt{Inventory}| = 2,000$;

The other statistics will not be necessary to answer the below question. In the below questions, for any equality predicate $R.A = v$ in a join or selection operator, assume that $1/|\Pi_A(R)|$ many tuples match irrespective of what v is. We will give an example in part (b). This is a simplified uniformity assumption to make cost and cardinality estimations in DBMSs simpler. We also assume the following DBMS setup:

- Each disk/memory block can hold up to 10 rows (from any table);
- All tables are stored compactly on disk (10 rows per block);
- 8 memory blocks are available for query processing.

Below, we ask you to describe a query plan and compute its I/O cost. You do not have to draw a plan as a tree. You can just describe the overall query processing algorithm that will execute and analyze its I/O cost.

(b) (8 points) Suppose that there are no indexes available at all, and records are stored in no particular order. What is the best execution plan (in terms of number of I/O's performed) you can come up with for the query

$$\sigma_{(\texttt{ShipDate}=\texttt{:today})}(\texttt{Order} \bowtie \texttt{Inventory})?$$

Note that this is a natural join, so the join is on the bookID column. For example, as per the uniformity assumption we made above, assume in your calculations that 1 in 1000 tuples in Order satisfy the shipDate=:today predicate because this is an equality predicate and $|\Pi_{ShipDate}(Order)| = 1000$. Describe your plan and show the calculation of its I/O cost.

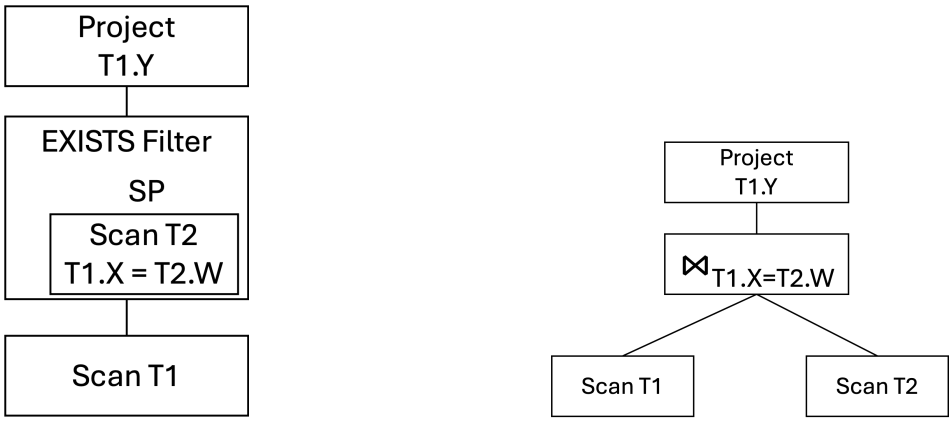
(c) (8 points) Suppose there is a B+-tree primary index on `Order`(`orderID`) and a B+-tree primary index on `Inventory`(`bookID`, `warehouseID`), but no other indexes are available. Furthermore, assume that both B+-trees have a maximum fan-out of 100 for non-leaf nodes; each leaf stores 10 rows; and all nodes in both B+-trees are at maximum capacity except the two roots. What is the best plan for the same query in (b)? Again, describe your plan and show the calculation of its I/O cost. Assume that all accesses to each B+ tree node (including the root) is 1 I/O cost (so do not assume that the root of the B+ is cached).

Question 3.

[9 marks in total] Consider the following two relations: T1 (X,Y) and T2 (W, Z), and the SQL query below.

```
SELECT Y
FROM T1
WHERE EXISTS (SELECT *
              FROM T2
              WHERE T1.X = T2.W)
```

Consider the two plans shown in Figure 1. The left plan P_1 scans each T1 tuple $t1_i$, passes this tuple to an operator called "Exists Subplan Runner" (ESR). ESR has an inner subplan SP that evaluates the inner sub-query in the original query. It passes $t1_i.X$ to SP. SP then scans $T2$ to find tuples of $T2$, whose W value are equivalent to $t1_i.X$. If any such tuple is found, $t1_i$ is passed to the next Project operator. Otherwise $t1_i$ is filtered out. This is repeated for each tuple in $T1$. $P2$ on the other hand *unnests* the query and turns it into a join-only query that joins $T1$ and $T2$ on $T1.X = T2.W$ and the output of the join is given to the Project operator. This optimization is called *subquery unnesting* (think about why Plan 2 looks like a better plan). For each of the below scenarios, argue if Plan 1 can be transformed into Plan 2, i.e., are the plans guaranteed to produce the same output. Answer T or F and provide a brief explanation.



(a) Plan 1 (b) Plan 2

Figure 1: Two plans for the SQL query above.

(a) (3 points) X is the primary key of T1. Answer T or F:

(b) (3 points) W is the primary key of T2. Answer T or F:

(c) (3 points) T2.W is a foreign key to T1.X (and T1.X is primary key). Answer T or F:

Question 4.

[22 points in total]

(a) (12 points) Determine whether or not each of the following four transaction execution histories is conflict serializable. If a history is conflict serializable, specify a serial order of transaction execution to which it is equivalent.

$$H_1 = w_1[x], w_1[y], r_2[u], w_1[z], w_2[x], r_2[y], w_1[u]$$

$$H_2 = w_1[x], w_1[y], r_2[u], w_2[x], r_2[y], w_2[y], w_1[z]$$

$$H_3 = r_1[x], r_2[y], w_2[x], r_1[z], r_3[z], w_3[z], w_1[z]$$

$$H_4 = w_1[x], w_2[u], w_2[y], w_1[y], w_3[x], w_3[u], w_1[z]$$

(b) (10 points) Consider the following sequence of operations, where,c_i and a_j are commit and abort requests for transactions T_i and T_j, respectively. Fore example, c_1 is the commit request for T_1 and a_4 is the abort request by T_4. Assume the database system uses strict 2PL protocol and follows the following lock acquisition and release rules:

- A transaction T attempts to acquire a lock on data item o, right before T is about to execute its first operation on o.
- A transaction T releases a lock on item o as early as possible ensuring that the transaction abides by the strict 2PL protocol.

Given the above rules, (i) indicate which, if any, of these requests will cause a transaction to block? (ii) Indicate if a deadlock will occur?

$$w_1[a], r_2[a], r_1[b], w_3[b], c_1, r_4[b], w_2[b], c_3, c_4, a_2$$

Part II

This part consists of one programming question. We have provided several starter files which you can find on Learn. These files are under A3/joa.zip file.

Submission

You need to submit your assignment via `https://marmoset.student.cs.uwaterloo.ca/`.

Marmoset will automatically test and mark your code. **IMPORTANT: Your submission must be a single zipped folder which includes only your `join_graph.py` file**. This means that when you unzip the folder you should see the `join_graph.py` file. Ensure that the file is named exactly `join_graph.py`.

Each question has public tests and private tests. You will only see the results of the public test (whether or not you passed or failed the public test).

Passing a public test does not guarantee that you pass the private tests. To receive full marks your code must pass all of the tests. Therefore, it is important that you test your code yourself.

To pass a test the output returned by your code must match the expected output. **Your code must also not exceed a timeout.** (A correct DP implementation will not come close to the timeout).

You can re-submit until the deadline and your best submission will be used for the final grade. Do not be alarmed if marmoset is slow, marmoset can often take a rather long time to run and grade your submission.

Setup

For this question you will be writing python. Specifically, python 3.

As with prior assignments, you can use your own machine or you can use the university's servers located at: linux.student.cs.uwaterloo.ca. You can remotely connect to these machines via ssh. If you are connecting from off campus you will need to have already setup ssh keys on the server. You can find detailed information about these servers here:
`https://uwaterloo.ca/computer-science-computing-facility/teaching-hosts`.

It is important to note that we cannot guarantee the uptime of the school's machines so we encourage you to install python on your own machine.

We have provided some starter files which can be found on Learn. Note that this starter code was written and tested in python version 3.10.12. Ensure that you are using a compatible version of python. These starter files include 2 python scripts: `main.py` and `join_graph.py`. You will need to edit the incomplete functions in `join_graph.py`.

For this question you must implement the `getCardinality` and `getBestJoinOrder` functions in the `join_graph.py`. You can add additional functions or variables in `join_graph.py` if you wish.

Do NOT edit the `main.py` file and do NOT edit the existing completed functions and variables in the `join_graph.py` file.

Question 5.

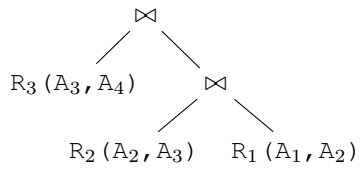
[30 marks in total]

You are required to implement a simple join optimization algorithm, JOA for short, for an RDBMS. To simplify our setting and optimization problem, we will assume that the input to JOA is a "natural chain join query Q":

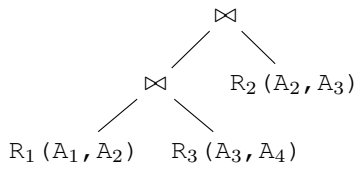
$$R_0(A0, A1) \bowtie R_1(A1, A2) \ldots R_{n-2}(A_{n-2}, A_{n-1}) \bowtie R_{n-1}(A_{n-1}, A_n)$$

We assume each join in $R_i(A_i, A_{i+1}) \bowtie R_{i+1}(A_{i+1}, A_{i+2})$ is a primary key-foreign key join and one of $R_i.A_{i+1}$ or $R_{i+1}.A_{i+1}$ is a foreign key to the other. Your goal is to find the lowest estimated cost join plan for Q. Each join plan is represented as a binary tree, where the leaves are the base tables R_i, each intermediate node o of the tree joins two base or intermediate relations I_ℓ and I_r, and outputs $I_\ell \bowtie I_r$. Note that this is equivalent to the output of all base relations, i.e. those at the leaves, of the sub-tree rooted at o. The root outputs Q, the join of all relations.
We give an example:

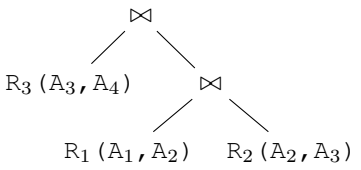
Example 1 *Consider the chain join of 3 relations R_1, R_2, and R_3 whose cardinalities are 10, 20, and 30 respectively. Three of the possible join trees are shown in Figure 2.*



(a) A valid join tree$_1$.



(b) An invalid join tree$_2$ because it performs a Cartesian product.



(c) Another invalid join tree$_3$ because R1, the smaller relation is on the left instead of right.

Figure 2: Possible join trees

In addition to being the lowest estimated cost join plan, your output should satisfy the following three constraints:

1. *Fullness:* Each relation R_i should appear once (and exactly once) in the leaves.
2. *Cartesian Product-free:* You should not be doing any Cartesian products anywhere in the tree, i.e., join a relation Int_ℓ and Int_r that don't share a common column. For example the tree in Figure 2b is not a valid output because the non-root intermediate node joins R_1 and R_3, which don't have a common column, so this is a Cartesian product. This effectively means that each intermediate join needs to be joining a consecutive set of relations from R_i to R_j, where $j > i$: $R_i \bowtie R_{i+1} \bowtie ... \bowtie R_j$.
3. *Right-relation is smaller property:* If an intermediate operator is joining Int_ℓ and Int_r, the (estimated) cardinality of Int_r, denote as $|Int_r|$, must be smaller or equal to the cardinality of Int_ℓ, i.e., $|Int_r| \leq |Int_\ell|$. For example the tree in Figure 2c is invalid because R2 has a greater cardinality but appears on the right side. The motivation for this constraint is that we assume that each join will be evaluated using the HashJoin algorithm. In this query processor, the right relation is the build side and the left is the probe side. Recall from the query processor lectures that a standard optimization for HashJoin's is to make the smaller table the build side.

If T is a join plan, the *cost of* T is the sum of the estimated cardinalities of its intermediate nodes, i.e., the sum of estimated cardinality of intermediate relations. For correctness of your algorithm, it will not matter if you include or exclude the root's cardinality in the cost (because it is a fixed cost no matter what sub-plans you pick) but is likely that your implementation will be simpler if you include it.

Example 2 *The plan in Figure 2a has only 1 intermediate node, which represents the join of $R1$ and $R2$. Suppose CardEst estimates that the cardinality of the join of $R1$ and $R2$ is 10. Then the cost of this plan would simply be 10.*

The goal of the assignment is to implement a dynamic programming (DP) algorithm that finds the lowest-cost plan, whose high-level description will be explained momentarily. We first give an overview of the setup in terms of input and initial source code.

Setup

Input File Specification

You specify a query by giving an integer n for the number of relations $R_0, ..., R_{n-1}$ followed by a set of cardinalities indicating the cardinality for each relation and a set of n-1 relations, suffixed by "R" which describe for each R_i, R_{i+1} which one has a foreign key to the other. So an example input could be:
4
5,10,15,20
R0, R2, R2

Here, 4 in the first line indicates that we are joining $R_0 \bowtie R_1 \bowtie R_2 \bowtie R_3$. The second line indicates $|R_0| = 5, |R_1| = 10, |R_2| = 15$ and $|R_3| = 20$. The third line "R0, R2, R2" indicates the following: (i) in the join $R_0(A_0, A_1) \bowtie R_1(A_1, A_2)$, $R_0.A_1$ has the foreign key to $R_1.A_1$; (ii) in the join $R_1(A_1, A_2) \bowtie R_2(A_2, A_3)$, $R_2.A_2$ has the foreign key to $R_1.A_2$; and (iii) in the join $R_2(A_2, A_3) \bowtie R_3(A_3, A_4)$, $R_2.A_3$ has the foreign key to $R_3.A_3$.

Initial Source code

You are given Python code in which you are supposed to implement the JOA in the empty method `getBestJoinOrder` belonging to a `JoinGraph` class. The method `getBestJoinOrder` returns a `JoinPlan` object representing the plan with the lowest cost. `JoinGraph` class also contains an empty `getCardinality` method that you are supposed to implement. The details of the given classes are as follows:

- `Relation`: Represents a relation in the input.
- `JoinPlan`: Represents a plan T and has a left and right child (which can be null) which represent the left and right sub-plans of a join operator. A plan also has `estOutCard` and `estCost` fields that respectively store the estimated cardinality of the output and the estimated cost of T. There is a `printPlan` function, which we will use to print your plans and check against our expected plans.
- `JoinGraph`: This is the main class that loads the description of a join query Q given an input file (using its `_load` method) and represents Q in memory as an array of relations $[R_0, ..., R_{n-1}]$ with an array of `JoinCondition` object.
- `JoinCondition`: Represents a primary-foreign key join condition with two relation object fields primaryRel and foreignRel where foreignRel refers the relation that has a foreign key to the primaryRel relation.

Cardinality Estimation

You are implementing `JoinGraph`'s `getCardinality` which takes a list of base relations that make up a chain join query and returns "CardEst". "CardEst" is the estimated cardinality of the relation obtained by

joining the input list of relations. The primary key-foreign key information are needed to produce consistent cardinality estimates. We estimate the cardinality of a (sub-) chain query $R_i \bowtie R_{i+1}...R_j$ as:

$$\frac{|R_i| \times |R_{i+1}| \times ... \times |R_j|}{F(R_i, R_{i+1}) \times F(R_{i+1}, R_{i+2}) \times ... \times F(R_{j-1}, R_j)}$$

where $F(R_t, R_{t+1})$ is $|R_t|$ if R_{t+1} has a foreign key to R_t or $|R_{t+1}|$ otherwise.

Consider computing the "CardEst" of the output chain query from Example 1, where suppose R_2 has a foreign key to R_1 and R_2 has a foreign key to R_3. Given the `Relation` objects R_1, R_2, and R_3, this would be obtained by calling `getCardinality`([R_1, R_2, R_3]). Then `getCardinality` would compute "CardEst" as follows:

$$\frac{|R_1| \times |R_2| \times |R_3|}{F(R_1, R_2) \times F(R_2, R_3)} = \frac{10 \times 20 \times 30}{10 \times 30} = 20$$

.

Note that in the above formula: a) $F(R_1, R_2) = |R_1| = 10$ because R_2 has a foreign key to R_1; b) $F(R_2, R_3) = |R_3| = 30$ because R_2 has a foreign key to R_3.

DP Algorithm

The JOA you are implementing is a DP algorithm to be implemented in `getBestJoinOrder` that goes through n iterations where n is the number of relations in the input chain query Q. Each iteration $i \in [1, n]$ generates the lowest cost plan that satisfies the 3 constraints above, for all possible subqueries of Q containing i *consecutive relations* (e.g., $R_k \bowtie R_{k+1} \bowtie ... \bowtie R_{k+i-1}$).

- As a base case of your DP algorithm in iteration $i = 1$, consider plans scanning each relation R_x which is a `JoinPlan` that consists of a single node that has the relation R_i.
- In iteration $i = 2$, for each $j, j + 1$, where $j \in [0, n - 2]$, compute the lowest cost join plan for queries containing two relations R_j, R_{j+1} that can be joined by joining two subplans computed in iteration $i = 1$. Do not forget to consider the *right relation is smaller property* when computing the best plan.
- ...
- In iteration $i = m$, for each $j, j + m - 1$, $j \in [0, n - m]$, compute the lowest cost join plan for queries containing m relations $R_j, R_{j+1}, ..., R_{j+m-1}$ that can be joined by joining two subplans computed in previous iterations. Iterate through each "splitting point" $k \in [0, m - 1]$, and consider the cost of joining the best plan P_{prefix} for $(R_i \bowtie ... \bowtie R_{i+k})$ and the best plan P_{suffix} for $(R_{i+k+1} \bowtie ... \bowtie R_{i+m-1})$. Note that you have already computed P_{prefix} and P_{suffix} in previous iterations, so you should be storing them in some data structure (e.g., a 2-dimensional array) and reusing them (and not recomputing them from scratch). Then out of these $m - 1$ possible plans, pick the best plan as the best possible plan to evaluate the join $(R_i \bowtie R_{i+1} \bowtie ... \bowtie R_{i+m-1})$. As before, do not forget to consider the "right relation is smaller property" when computing the best plan.
- And so on and so forth till iteration $i = n$, in which you compute and return the lowest cost plan for Q $((R_- \bowtie R_1 \bowtie ... \bowtie R_{n-1})$.

You can write your algorithm in a recursive or iterative manner. Both will be accepted.

[**Note 1**: Do not change the methods or the fields in the given classes. The tests rely on the given implementation. You can add fields or helper methods as necessary to use to implement `getCardinality` and

`getBestJoinOrder`.]

[**Note 2**: There are both private and public tests. There are 12 public tests and 18 private tests. Tests are constructed based on queries and for each query, we have two tests: i) test that `getCardinality` returns the correct estimates for every possible subquery chain; and ii) test that `getBestJoinOrder` returns the lowest cost plan. Cardinality estimates tests are meant to help you with debugging and are worth 0 marks. The join order tests are worth 2 marks each.]

[**Note 3**: If you implement a brute force algorithm that checks all join plans (and there are **a lot of** plans even in queries with 7 or 8 relations!) your algorithm will likely timeout. So do implement a DP algorithm as described above. We are asking for a dynamic programming algorithm. Our goal is to expose you to a simplified version of an actual algorithm that is really implemented in many DBMSs.]

[**Note 4**: If multiple join orders have the same minimum cost, you can return any of them.]