PG Software Lab - CSP 509

Major-project Specification Phase B

Due date: Nov 22, 2019 11:55pm

Total Weightage 12%

Instructions:

- All submissions must be made through the Moodle site for this course
- Only one submission per team would be considered and graded. It would be assumed that all members of the team have participated equally and same score would be given to all members of the team.
- Your submission should have names of all the members of your team.
- Any assumptions made while solving the problem should be clearly stated in the solution.
- **You are not allowed to different teams across phase A and phase B of the project.**
- As always correctness of the algorithm must be ensured.
- TAs would be quizzing you on your code. You must understand each and every line of your submitted code. Also the implementation specifications mentioned in the questions need to be strictly followed. Failure to adhere to these requirements would result in substantial loss of points.
- Also pay attention to the scalability of the code. Your code would be tested on large datasets.
- Very Important: Your code should not have a directory structure. All files (code + dataset + written material for questions) should be present in just one folder. Note that this is absolutely crucial for grading this assignment.

This phase consists of the following major tasks: (a) Implementation of a secondary memory based index structure (**SI**) over the file (split on blocks) containing the sales transaction records described in phase A of the project; (b) build a range query algorithm which can use **SI** for answering the range queries; (c) Adapt the extendible hash index structure to answer the range queries; (d) implement a naive algorithm to answering the range queries. Following this, you are required to construct a cost based query optimization routine. Given a range query, this route, would determine the most optimal way of answering the given range query.

Sample Range Query:

Select * From Transaction Where TID is between <X> and <Y>

Implementation Specifications

Secondary Memory based index structure (Refer to class notes, following is just a summary): This is multi-level index structure stored across a collection of "disk-blocks." Assume that **gamma** number of index records can fit in each "disk-block" As done in phase A, these "disk-blocks" are actually files on your hard drives. In your implementation, you can create this index structure before hand and store in your hard drive. Following is a brief description of the levels of this index structure:

Level 0 (leaf level): For each record in the transaction file, include a record **{TransactionID, <Filename of the disk block which stores this record>}** in the level 0 of the index. In other words, for every record in file, we would have a corresponding record in the level 0 of this index. These records are stored in files simulating "disk-blocks" containing **gamma** records each. All "disk-blocks" in this level are linked, ie., each file simulating the "disk-block" would have the name of the previous and the next "disk-block" in this level. It is important to note that index records of this level are sorted (on transaction ID) across the "disk-blocks." This property is very important to ensure the correctness of the range query.

Level 1: Using the previously described procedure, level 0 would span across several "disk-blocks." We pick **the first record** from each of the level 0 blocks to create entries in level 1. Note that records in this level would be of the form:

{TransactionID, <Filename of the disk block which stores this entry in Level 0>} Similar to level 0, the records of this level are also stored in files simulating "disk-blocks" containing **gamma** records each. And once again, index records stored in this level are sorted (on transaction ID) across the "disk-blocks." This property is very important to ensure the correctness of the range query.

..

..

..

Continue until all records fit in just one "disk-block"

Implement a correct and efficient range query algorithm on this index structure. For our given sample range query, this algorithm should first search for the smaller transaction id "<X>" in the index structure and then perform a leaf scan and retrieve all the corresponding records from the transaction file. No other range query algorithm would be accepted for secondary memory index structure.

Cost Modeling:

Given a range query **R,** the cost of a particular range query algorithms is defined as the estimated number of "block" transfers made while answering the query **R.** Use the cost expressions covered in the lecture for calculating the estimated number of blocks transferred. Note that in case of extendible hash, the number of blocks transferred would correspond to the number of "buckets" read from the "simulated secondary memory." Whereas, in the case of secondary memory index and the naive algorithm, the number of blocks transferred would be number of "files simulating disk-blocks" accessed.

Query Optimization Routine:

Given a range query **R,** the goal of this routine is to determine the most efficient range query algorithm (amongst naive, extendible hash and secondary memory index) for this particular range query. For this purpose, the routine uses the cost expressions (covered in the lecture) of different algorithms and chooses the algorithm with the least cost. Recall that the expressions internally use the parameter **theta** which denotes the number of records which are likely to be present the given range between **<X>** and **<Y> . For estimating theta, you should use a histogram built over the dataset.**

<u>Experimental Analysis</u>

Compute the following for each of the following queries: (a) estimated number of blocks for naive search, extendible hash and secondary memory index; (b) choice of your query optimizer and actual blocks accessed by your chosen range query algorithm; (c) Actual number of blocks accessed by each of the other range query algorithms.

Assume the following parameters:

1. Create 1.5 lakh records in the transaction file.
2. The transaction IDs are random (unique) integers between 50,000 and 350,000.
3. While storing the transaction file in "disk-blocks" — Store 100 records per "disk-block"
4. 1 bucket in "simulated secondary memory" of extendible hash can store
    - 200 records of the form {TransactionID, <Filename of the disk block which stores this record>}
    - 300 entries of the bucket address table
5. **Gamma** — 200 index records for the secondary memory index "disk-blocks" (of the form {TransactionID, <Filename>})

Query 1: Select * From Transaction Where TID is between 50,000 and 51,000

Query 2: Select * From Transaction Where TID is between 50,000 and 55,000

Query 3: Select * From Transaction Where TID is between 50,000 and 60,000

Query 4: Select * From Transaction Where TID is between 50,000 and 70,000

Query 5: Select * From Transaction Where TID is between 50,000 and 90,000

Query 6: Select * From Transaction Where TID is between 100,000 and 180,000

Query 7: Select * From Transaction Where TID is between 100,000 and 260,000

Query 8: Select * From Transaction Where TID is between 100,000 and 300,000

Query 9: Select * From Transaction Where TID is between 100,000 and 350,000

Extra Work for team-size 3 (please meet the instructor once in this regard)

Develop a "disk defragmenter" routine for your extendible hash. This routine is supposed to clean and reclaim any unused buckets in the "simulated secondary memory".