Database System Implementation- CSE 507

Homework 1

Due date: January 28, 2017 9:00am

Instructions:

- All submissions must be made through usebackpack site for this course (https://www.usebackpack.com/iiitd/w2017/cse507)
- 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.
- Only one submission should be uploaded per team.
- Any assumptions made while solving the problem should be clearly stated in the solution.
- 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.
- Question 2 is for teams of size 3. This questions will not be graded for teams for size 2 or less.
- **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.

Question 1 (Programming Question) (120 points)

This question would require you to implement and compare the performance of Extendible Hashing and Linear Hashing. You may use any high-level language to implement these structures. Java or C++ would be most preferable. Implementations using statistical packages like Matlab or R will not be considered. Specialized libraries for managing hash tables must not be used. You may existing libraries for basic data structures like vectors and associative array (map in C++ STL libraries). You may also use math libraries as needed for tasks like, generating random numbers, converting to and from binary format, etc.

Implementation specifications:

You may choose to implement the required structures in main memory, but secondary memory behavior must be simulated. Following tips would help you achieve this.

(a) The secondary memory can be simulated through a vector of fixed length arrays. Here, each of the arrays would be a bucket (a block) and the indices of the vector would be taken as the bucket address (block address).

(b) Apart from records, each bucket should also have information on number of empty spaces and an index (in the vector) containing the next bucket of the file (or the overflow chain).

(c) The last bucket of the file (or the overflow chain) must have a special character denoting that it is end of the file (or the overflow chain).

(d) It is advisable to keep a separate area in the vector for storing the overflow buckets.

(e) Main memory is another vector of fixed size and is not dived into buckets. Note that you would need to simulate main memory only in case of Extendible hash.

(f) There should be only one instance of secondary memory running in your code.

Information on File Records

(a) All records are of fixed size. For this homework, each record would be a single integer between 0 and 800000.

(b) The bucket capacity is fixed in terms of number of records it can contain, i.e., a bucket is basically a fixed number of records (integers). You may ignore the space requirements of storing the next bucket in case of overflow chain. The bucket capacity mentioned in the question is just the number of data records that can be stored in the bucket (over and above any other bookkeeping needed).

(c) Expansion takes place as soon as a bucket overflows.

Extendible Hash

(a) The most significant bits are extracted to find the directory entry.
(b) Only one split per record insertion. In other words, we would not try a second split even if the first split fails to release the overflow bucket or ends up again in an overflow situation. In this case, we will just have an overflow bucket.
(c) "Main memory" can hold upto 1024 directory entries. The rest resides in "Secondary memory."

Linear Hash

(a) A simple division with modulo arithmetic is used to find the relevant bucket, i.e., it will not have a directory similar to Extendible hash.
(b) If needed you add or subtract a fixed number from the result of the modulo arithmetic to map it to appropriate index in the vector simulating the secondary memory.

Datasets to be created:

(a) Dataset-Uniform: Contains 100000 uniformly distributed random numbers between the range of 0 and 800000. Numbers may get repeated.
(b) **Dataset-HighBit:** Contains 70000 numbers uniformly generated between the range 700000 and 800000, and 30000 generated between the range 0 and 700000. Numbers may get repeated.

Experimental Analysis:

Goal: The performance of Extendible Hashing and Linear Hashing needs to be compared in terms of number of "disk accesses" for insert and search operations on these hashing techniques.

Parameters used:

(a) **N:** The number of records currently in the hash table.
(b) **B:** The number of buckets current in the hash table + Number of buckets in the overflow.
(c) **b:** Bucket capacity
(d) **bs**: The number of buckets accessed for a successful search (+1 if the directory is not in the main memory in case of extendible hash)
(e) **s:** Number of successful searches

Comparison Metrics

(a) Storage utilization: N/(B*b)
(b) Average successful search cost: bs/s
(c) Splitting cost:

Linear Hash: 1 access to read the bucket to be split + k accesses to read k overflow buckets (if any) + extra accesses to writ the overflow buckets (if any) attached to new and old buckets

Extendible Hash: k access to read overflow buckets attached to the bucket being split (if any) + 1 access to write the old bucket + 1 access to write the new bucket + extra accesses to write the overflow buckets attached to old and new buckets + accesses needed to update the directory pointers if the directory resides on "secondary memory."

Experiment to be conducted

(I) Keep inserting records from the dataset you created and continuously measure the values of the metrics (a) and (c). While metric (a) is perfectly continuous in nature, metric (b) will gather data-points only you see a split.

(II) After every 5000 records randomly generate 50 search queries and evaluate the metric (b)

Repeat these for both the datasets you created and for each dataset and plot the following three Plots for each dataset:

Plot 1: **Metric (a) against the number of records in the file for both Linear and Extendible hash for Bucket size 10 and 70.**

Plot 2: **Metric (b) against the number of records in the file for both Linear and Extendible hash for bucket size 10 and 70.**

Plot 3: **Metric (c) against the number of records in the file for both Linear and Extendible hash for bucket size 10 and 70.**

Deliverables for Question 1

- **(A) Implementation code for Linear and Extendible Hash**
- **(B) Brief text on the implementation logic for the algorithms. You should include documentation on the code submitted.**
- **(C) The required 6 plots**
- **(D) A brief explanation of the trends (cross over points and general trends) observed. This should be a non-trivial explanation. You should interpret the trends in terms of the properties/behavior of the algorithms.**
- **(E) Datasets used in the experiment.**

Question 2 (Extra question which is compulsory for teams of size 3) (40points):- For this question you need to design and implement the delete operation in Linear Hashing. Delete operation has two parts: (1) search of the record to be deleted; (2) delete the record from the bucket. If deleting causes an underflow (i.e., the bucket gets empty) then we need to call a *merging algorithm* which merges two buckets (you need to figure out which ones) in case of an underflow. You need to design the logic of the merging algorithm for this question.

You don't need to run any experiments on the delete and the merge algorithm. The TAs would just check the correctness of the code through some test cases. Please submit a pseudocode of the merge algorithm along with your submission. In your pseudocode, you may use global file parameters like N (next to split) and value if I (corresponding to hash key family *key mod 2^I*). Note that your merging algorithm should be such that it works alongside splitting algorithm smoothly. You may consider the following scenario. Records are inserted and deleted from the file, whenever there is an overflow we call the ``splitting algorithm'' and whenever there is an underflow we call the ``merging algorithm.'' Overall there should be consistency, i.e., values of N and I should be correct and records should be in their proper buckets. **Hint:** Merging algorithm is essentially an opposite of the splitting algorithm in Linear Hashing.

For this question, you need to submit the following:

- (a) Code for Linear Hash which has delete operation (and merge operation) implemented in it.
- (b) A pseudocode of the merge algorithm as a text file.